CDOSim: Simulating Cloud Deployment Options for Software Migration Support

Florian Fittkau, Sören Frey, and Wilhelm Hasselbring
Software Engineering Group
Kiel University
24118 Kiel, Germany
{*ffi, sfr, wha*}*@informatik.uni-kiel.de*

***Abstract*—The evaluation of competing cloud deployment options (CDOs) forms a major challenge when migrating software systems to cloud environments. For example, there exists a plethora of potential cloud provider candidates, components must be mapped to suitable virtual machine instances, and, to exploit elasticity, appropriate runtime adaptation strategies for specific usage profiles have to be defined. But analyzing potential CDOs manually is intractable, costly, and time-consuming due to the heterogeneity of the cloud environments and the overall combinatorial design space complexity.**

We present the simulation tool CDOSim that can simulate cost and performance properties of those CDOs. It builds upon and significantly extends the cloud simulator CloudSim and integrates into our cloud migration framework CloudMIG. Additionally, we created a cloud benchmark to augment CloudMIG's cloud environment models with provider-specific performance characteristics. Along with this simulation input, CDOSim utilizes reverse-engineered architectural models and can employ actual monitored workload. We report on extensive experiments incorporating Eucalyptus and Amazon EC2 which show that CDOSim can sufficiently accurate predict the cost and performance properties of CDOs.

***Keywords*-cloud deployment options, CDOSim, simulation, cloud migration, cloud benchmark.**

I. Introduction

Cloud computing is emerging as a promising new paradigm that aims at delivering computing resources and services on demand. Finding the best suited cloud deployment option (CDO) during a migration is a complex task [1]. The most obvious and basic CDO is the selection of a cloud provider. Due to the heterogeneity of current cloud environments, this choice can have a significant impact on the deployed application's performance characteristics and operational expenditures [2]. Furthermore, the mapping between services and virtual machine (VM) instance types and quantities must be considered and the specific adaptation strategies, like allocating a new virtual machine instance if the CPU utilization is above a given threshold, have to be chosen and configured, for instance. The set of combinations of the given choices forms a huge design space which is infeasible to test manually. Simulating a CDO can assist in solving this problem.

We present the simulation tool CDOSim [3] that can simulate the response times, SLA violations, and costs of a CDO. Besides these core simulation capabilities, CDOSim is designed to address the major shortcomings of other existing cloud simulators in the context of simulating CDOs:

(1) It is consequently oriented towards the cloud user perspective instead of exposing fine-grained internals of a cloud platform when following the cloud provider perspective. (2) We developed an accompanying benchmark that mitigates the cloud user's lack of knowledge and control concerning an underlying cloud platform structure and it's impact on an application's overall performance. (3) CDOSim uses application models that follow the Knowledge Discovery Meta-Model (KDM) [4] of the OMG. Hence, the simulation is independent of concrete programming languages in the case appropriate KDM extractors exist for a particular language. Therefore, application models often can be created using automated reverse-engineering techniques. (4) Workload profiles from production monitoring data can be used to replay actual user behavior for simulating CDOs.

For these purposes, we built upon and substantially extended the cloud simulator CloudSim [5]. CloudSim employs the notion of million instructions per second (MIPS) to model computational complexity and the capacity of virtual machine instances. However, it lacks a description of practicable derivation mechanisms. We refine the MIPS unit to the mega integer plus instructions per second (MIPIPS) unit. Our benchmark calculates a MIPIPS score and weights that allow to derive MIPIPS values for other instruction and data types. Furthermore, CDOSim integrates in our cloud migration framework CloudMIG [6]. The corresponding tool CloudMIG Xpress [1] incorporates CDOSim as a plug-in.

We conducted extensive experiments using our private cloud Eucalyptus and the public cloud Amazon EC2. The evaluation shows that CDOSim can sufficiently accurate predict the cost and performance characteristics of CDOs.

The remainder of the paper is structured as follows. Section II describes the fundamentals. Then, Section III introduces the MIPIPS and weights benchmark. CDOSim is presented in Section IV in conjunction with the simulation inputs and outputs. Afterwards, Section V evaluates the benchmark's and CDOSim's accuracy, before the related work is described in Section VI. The final Section VII draws the conclusions and outlines the future work.

[1] http://www.cloudmig.org, last accessed 2012-07-05

II. Fundamentals

This section lists basic concepts and used technologies.

A. Cloud Deployment Option

We define the fundamental concept of a CDO as follows:

Definition 1: In the context of deploying software on a cloud platform, a cloud deployment option (CDO) is a combination of decisions concerning the selection of a cloud provider, the deployment of components to a number of virtual machine instances, the virtual machine instances' configuration, and specific runtime adaptation strategies.

The deployment of components to virtual machine instances includes the possibility of forming new components of parts of already existing components. By a virtual machine instances' configuration, we refer to the instance type of virtual machine instances. For example, the *m1.small* instance type in the case of Amazon EC2, Furthermore, an example for a runtime adaptation strategy is "start a new virtual machine instance when for 60 seconds the average CPU utilization of allocated nodes stays above 70 %."

B. CloudSim

We build upon and significantly extend CloudSim [5], which is a cloud computing system and application simulator. It supports modeling and simulating large cloud computing environments with a single computer and is a self-contained platform for modeling clouds, the service brokers, and different policies for resource allocation, for instance. CloudSim also provides support for network connection simulation between nodes and offers a facility for the simulation of a federated cloud environment. CloudSim has been successfully used by other researchers for simulating task scheduling in the cloud or power aware cloud computing.

C. MIPIPS

CloudSim requires MIPS as a measure for the computing performance of virtual machine instances. However, we consider MIPS values as too coarse grained. Most CPUs need different durations for different low level instruction types. For example, a division of two double values typically takes longer than an addition of two integer values on current CPUs. Furthermore, CloudSim does not specify the intended instruction type and how to measure the MIPS.

We introduce MIPIPS as the measure for describing the computing performance and express instructions like double plus as integer plus instructions through a conversion. Notably, we could have used, for example, mega double plus instructions per second (MDPIPS) as the measure for computing performance and normalized all other instructions to double plus instructions (see III-B for details).

We do not utilize existing software profiling methodology because we wanted an easy to adapt benchmark and a measure that correlates with the executed static statements.

D. Weights per Statement

An instruction counting approach that is described later in Section IV-C1 bases on static analysis and requires weights for different statements such that it can convert the instruction count (IC) of a statement into an integer plus IC. By measuring the times an integer plus instruction and, for example, a double plus instruction consumes, we can approximate that in the period of time of one double plus instruction the CPU could have performed x integer plus instructions. For instance, we measure 5 nanoseconds for a double plus instruction and we measure 2 nanoseconds for an integer plus instruction. Then, in the time where one double plus instruction took place 2.5 integer plus instructions could have been performed.

III. MIPIPS and Weights Benchmark

Our benchmark provides the calculation of MIPIPS and weights for different programming languages. Virtual machine instances typically provide only command line access. For this purpose, the interaction can be handled through a console as well as through a GUI. In the following the derivation of MIPIPS and weights per statement is described.

A. MIPIPS

1) Derivation: The basic idea for deriving MIPIPS is a benchmark that measures the runtime of a defined amount of integer plus instructions.

We use a loop which runs our integer plus instructions at least for ten seconds on current CPUs. Measuring the runtime of the whole loop would include more instructions like jumps and comparisons being measured. Therefore, we conduct a calibration run (see Listing 1 without the bold statements) for running the loop and then do a second run with our integer plus instruction added (see Listing 1 with the bold statements) to the loop's body. Afterwards, we subtract the runtime of the first run from the runtime of the second run. This reveals the execution time of the added integer plus instruction.

Our runtime measuring technique is a program that acts as a master and starts the benchmark run in a slave on the same machine. The runtime measurement is conducted by the slave program due to exclusion of initialization time. After the execution, the slave returns the measured runtime for the benchmark run to the master. According to [7], this measurement must be conducted at least 30 times. The number of runs can be configured by parameters or from a GUI. Afterwards, the master calculates the median of the response times.

An important part is the disablement of optimizations for the compiler and interpreter when the slave program is called by the master program. Depending on the selected language and optimization settings, the optimization could cause our loop to have constant runtime.

Listing 1. Calibration run (without bold statements) and MIPIPS counter (with bold statements) in Java

```
1  int x = 0;
2  int y = 0;
3  long start = System.currentTimeMillis();
4  int i = -2147483647;
5  while (i < 2147483647) {
6    x = x + 2;
7    y = y + 3;
8    i += 1;
9  }
10 long end = System.currentTimeMillis();
11 long difftime = end - start;
12 System.out.println(difftime);
13 System.out.println(x);
14 System.out.println(y);
```

2) Generation: For supporting easy adaptability for new programming languages, we utilize Xpand [2] to generate the benchmark for different target languages. Xpand requires a meta-model definition, an instance of the meta-model, and a language-specific generation template.

We developed an Ecore-based meta-model for the representation of a benchmark class. It contains the basic elements of an imperative programing language and enables the modeling of classes, methods, expressions, variable declarations, loops, class creations, and concrete method calls. Two special classes are included in the meta-model. These are *SystemOut* which represents the statement for printing Strings to the console and *MeasureTime* which represents the statement for getting the current value of a time counter. These two classes are mapped by the generation template to individual statements for each target language and can be quite different like `System.out.println()` for Java and `puts` for Ruby.

If a new language shall be supported by the whole benchmark, the programmer simply has to add a generation template for the desired language. He does not need to change the declarations of each benchmark. Currently, we support the languages Java, C, C++, C#, Python, and Ruby.

B. Weights per Statement

1) Derivation: For deriving the weights, we utilize the same approach which was described in Section III-A1 for measuring the MIPIPS. Instead of running a MIPIPS counter, the approach calculates the mega instructions per second for each statement, e.g., for double minus. Afterwards, it divides the MIPIPS value by the corresponding mega instructions per second for each statement.

Our benchmark program contains a set of weight benchmarks for the most used statements. The considered data types are integer, float, double, long and for each the operations plus, minus, divide, multiply. Furthermore, benchmarks for boolean and, boolean or, boolean not, class creation, field access, function call, and String plus are available. More weight benchmarks can be added by creating language-independent instances of our class definition meta-model to the benchmark generator and using Xpand for the generation of the source code.

[2]http://wiki.eclipse.org/Xpand, last accessed 2012-07-05

IV. The Simulator CDOSim

CDOSim can simulate cloud deployments of software systems that were reverse-engineered to KDM models. This section gives an overview of basic components and the integration with CloudMIG Xpress, presents the general simulation approach, describes the simulation input and output, as well as our CloudSim enhancements.

A. Overview

The integration with CloudMIG Xpress is depicted in Figure 1. Here, CloudMIG Xpress' cloud profiles represent cloud environment models. They are enriched with results from benchmark runs that were executed on virtual machines of the selected cloud environment and can be reused in several cloud migration projects. Here, a central online repository is planned for future work. A CDOSim user has to execute the benchmark on his status quo computing infrastructure (or a similar system) where the regarding software system is deployed. Monitoring data and extracted KDM models are also used by CloudMIG Xpress to build workload profiles and mapping models, respectively. A mapping model determines, for example, which KDM code models have to be deployed on specific virtual machine instances as well as the costs of those instances. CDOSim utilizes the mapping models and workload profiles for its simulation. Here, three basic instruction counting mechanisms are employed, they are described further in Section IV-C1.

B. Simulation approach

Five activities are conducted for a simulation. The first activity is *IC derivation*. In this activity, CDOSim conducts the derivation with one of the IC mechanisms that are described in Section IV-C. The derived ICs are written as attributes into the KDM instances that were wrapped by the mapping model and passed to the simulation. Secondly, the size of included types is derived. We need the type size to approximate the bandwidth that is used when there are distributed calls to other virtual machine instances. The next activity is the transformation from the mapping model provided by CloudMIG Xpress to our extended CloudSim meta-model. Then, the actual simulation with CloudSim takes place. Finally, the new simulation result is rated relative to the other runs.

Our approach assumes that approximately constant computing resources are available to virtual machines. Hence, it is not directly applicable for cloud providers that allow regular CPU bursting when resources are available.

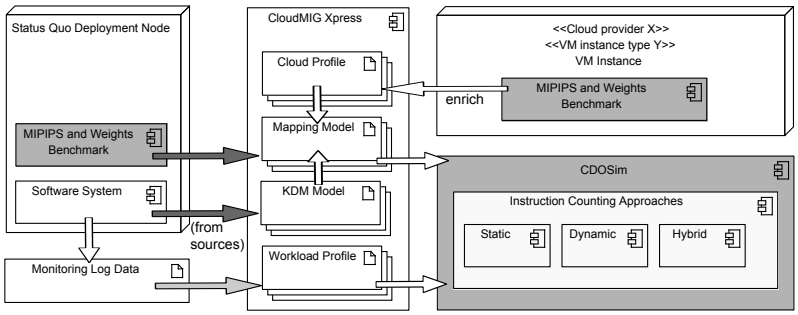


Figure 1. Integration of CDOSim with CloudMIG Xpress. The CDOSim and benchmark components are colored gray. Dark gray arrows indicate basic data the user needs to provide for simulating CDOs. Data marked with light gray arrows is only needed for dynamic and hybrid instruction counting.

C. Simulation Input

The simulation requires a mapping model that contains the previously described MIPIPS value and weights per statement measured by our benchmark as well as the actual KDM code deployment model. Furthermore, it needs the IC for each method call. The derivation method for the IC is illustrated in the following.

1) Instruction Count: The IC is required as a representation of the work that must be conducted for a call to a program or web service. For approximating the different inputs for each method, we utilize a mean value of instructions that should be executed. In combination with the MIPIPS, the IC approximates the runtime on a computer with the corresponding MIPIPS. For example, assuming 2 MIPIPS and 100,000 instructions for a call to a web service, the runtime of the call to the web service will be approximately 50 milliseconds.

We consider different possible preconditions for the derivation on which a specific approach can be chosen. The first approach, named *dynamic approach*, requires an instance of the code package of KDM, results of a dynamic analysis with contained response times, and the MIPIPS of the computer where the dynamic analysis took place. It utilizes the method definitions in the KDM instance and response times from the dynamic analysis. The second approach, named *static approach*, requires instances of the code and action package of KDM. The action package contains statements like condition blocks. The contained statements are counted in the *static approach*. Both approaches have shortcomings which we address in our third approach, named *hybrid approach*, which requires the preconditions of both the *dynamic approach* and *static approach*.

Dynamic Approach: One precondition of the *dynamic approach* is the availability of response times from a dynamic analysis, i.e., response times have to be monitored at runtime of the software under study. Optimally, there exists a phase of only low CPU utilization in the dynamic analysis such that the response times result from the execution of the method's instructions without major scheduling effects. Furthermore, the MIPIPS of the computer, where the dynamic analysis took place, have to be available. First, the *dynamic approach* computes the median response times during phases of low CPU utilization for each method in the workload. The IC of each method in millions is calculated by multiplying the median response time with the MIPIPS of the platform that measured the response time. For example, assuming the median of response times is 20 milliseconds and the MIPIPS is 200. Then, the resulting IC in millions is $0.020 \cdot 200 = 4$.

Notably, often not all methods are contained in the monitored workload. These methods get a -1 IC which indicates that those methods have no IC and that an error must be thrown when they are accessed in the simulation.

Static Approach: The *static approach* requires KDM instances that contain the code and action package of the software under analysis, but no monitoring data. It iterates over all methods and counts the instructions, according to defined equations for each control flow element, in an accumulator variable for each method. Then, the IC is annotated to each method. When no monitoring data is available, a synthetic workload profile can be defined using CloudMIG Xpress that is then used for the simulation.

Hybrid Approach: This approach combines the advantages of the *dynamic approach* and *static approach* and thus rules out some disadvantages of the former ones. The *dynamic approach* considers the real runtime of a method and thus can predict the runtime of a method very closely. However, it cannot derive an IC for every method because data from dynamic analysis is often incomplete. The *static approach* often results in wrong ICs because it does not consider optimization from the compiler, for instance. However, it can derive an IC for every method.

To combine the advantages, we take the ICs from the *dynamic approach* and try to correct the ICs from the *static approach*. Furthermore, the missing values for the IC of methods in the *dynamic approach* are filled with the ICs from the *static approach*.

D. Simulation Output

At first, we describe the different output components. The outputs for different cloud deployment options have to be comparable to propose which deployment option is more suitable. Hence, we describe our rating approach at the end of this section.

1) Cost: The first output of the simulation are the costs of the simulated cloud deployment option. It represents an overall cost which is the sum of the costs for the used bandwidth and virtual machine instances.

2) Response Times: The second output of the simulation is the median of the response times for each called method. Each method is then rated by our rating approach. Based on the method ratings, an overall rating for the response times is calculated.

3) SLA Violations: The third output is the number of violations for each SLA. Currently, only the SLA "a call will not timeout" is implemented. The GUI provides the possibility to change the timeout value. However, we plan to support a generic definition and processing of SLAs.

4) Rating: For comparing different simulation runs that cover different cloud deployment options, we conduct a rating for each run. The rating scale ranges from 1 as the best to 5 as the worst performance. Our rating approach searches for the best performance of all runs in each category and sets this performance as 1. The same is done for the worst performance and the corresponding run is set as 5. The other runs are rated relatively to the best and worst run with the following method. The median of all runs is calculated and set as 3. The same is conducted for the left and right set formed by the median of the whole set. Then, unassigned values are approximated linearly.

The overall rating forms a combined rating by multiplying each output rating with a weight, summing the resulting values, and rounding it at the second decimal place. It can be configured which output is more important by specifying other weights than the default weight of 0.33 for each output.

E. CloudSim Enhancements

CloudSim simulates cloud systems following a cloud provider perspective and uses so called Cloudlets to model abstract computations. However, we require the cloud user perspective for the simulation of CDOs. Hence, we extended CloudSim by the required features. These features range from the implementation of a CPU utilization model that is not basing on randomness to a new Cloudlet scheduler that enables Cloudlets to *call* other Cloudlets. Furthermore, virtual machines can be started and stopped during runtime.

V. Evaluation

We conducted three evaluations which we name E1 to E3. E1 evaluates the validity of the MIPIPS benchmark. The MIPIPS value should correlate with the performance of the underlying node. Furthermore, it should stay approximately constant on the same node. The MIPIPS value is important when there is a conducted run on a cloud provider or a local server and we want to simulate which performance and costs the workload would induce regarding a specific cloud provider. Due to space limitations, we have to restrict the evaluations incorporating the instruction counting approaches in E2 and E3 to the *dynamic approach*. Detailed analyses covering the *static approach* and the *hybrid approach* can be found in [3]. In E2, the validity of the simulation results in a comparison to real, measured runs on Eucalyptus and Amazon EC2, that were conducted with single core instances, are evaluated. This evaluation is important to assess the basic validity of the simulation. Evaluation E3 combines E1 and E2. It evaluates whether the prediction of the performance of a cloud provider basing on the data from another cloud provider is sufficiently accurate.

Following the three types of validity for simulation models by Zeigler [8], E2 evaluates the replicative validity and E3 addresses the predictive and structural validity.

First, we describe our methodology. Afterwards, the basic experiment setup is shown. Then, E1 to E3 are illustrated.

A. Methodology

For determining if the MIPIPS stay approximately constant on the same platform, we calculate the mean value and the standard deviation over the MIPIPS values resulting from the conducted runs. Then, we compare the standard deviation to a predefined threshold. The correlation between the performance of the underlying host and the MIPIPS is evaluated by comparing the MIPIPS with the available performance attribute and checking if the difference between the MIPIPS values is lower than a given threshold. The mean MIPIPS value should only differ by a small factor, which is lower than 2.5 %, if the benchmark is conducted again on the same instance type. The 2.5 % are motivated by the typical alpha level of $\alpha = 0.05$. For the evaluations E2 and E3, we compare the simulated values with the measured values per minute. The values are CPU utilization, instance count, costs, and response times. The following metric describes the relative error for each aspect of the simulation. All percent values will be truncated after the second decimal place.

Let T be the set of all minutes in the measurement duration. Let $m(t)$ be the measured value at timestamp $t \in T$ and $s(t)$ the simulated value at timestamp $t \in T$. When $m(t)$ equals 0, t is removed from the set T. (1) shows the formula to calculate the relative error for a timestamp t.

$$re(t) = \frac{|m(t) - s(t)|}{m(t)},\ m(t) \neq 0,\ t \in T \tag{1}$$

$$RE = \frac{\Sigma_t\ re(t)}{|T|} \tag{2}$$

(2) displays the formula for the calculation of the relative error (RE) for the whole simulation run. We feature four different REs. RE_{CPU} is the relative error of the CPU utilization. RE_{IC} stands for the relative error of the VM instance count. RE_{Costs} marks the relative error of the costs output. RE_{RT} is the relative error of the response times.

$$OverallRE = \frac{RE_{CPU} + RE_{IC} + RE_{Costs} + RE_{RT}}{4} \quad (3)$$

To enable a consolidated comparison between the results, we introduce the overall relative error ($OverallRE$), which is shown in (3). $OverallRE$ should remain below 30 % to have results that are sufficiently accurate [9].

B. Basic Experiment Setup

This section describes the common setup for our evaluations. We use iBatis JPetStore 5.0[3] for the evaluations E2 and E3, which is a web store for pets. Also for E2 and E3, new VM instances are spawned when the average CPU utilization of all allocated nodes is higher than 70 % for at least 1 minute. VM instances are terminated if it is below 30 % for at least 1 minute.

1) JPetStore Adaptation: Most calls to JPetStore are processed in less than 2 milliseconds, which results in only a small CPU utilization. In our evaluation, we want to use capacity adaption based on CPU utilization. Hence, we decided to generate additional CPU utilization for each call.

2) Amazon EC2: This section describes our deployment and utilized cost model for the experiments conducted on Amazon EC2.

Deployment: Our experiment environment incorporates one node, denoted the SLAstic node, and a Tomcat image. In the Amazon EC2 cloud, instances of the Tomcat image can be started. The Tomcat image includes Tomcat 6.0.18 with JPetStore 5.0, its own database, and the monitoring framework Kieker 1.4.[4] Kieker is included for monitoring the CPU utilization and response times of the annotated methods in JPetStore. It sends the monitored data to an ActiveMQ 5.5.1 queue on the SLAstic node. SLAstic [10] analyzes the CPU utilization obtained from the ActiveMQ queue and calculates if a new instance of the Tomcat image should be allocated or if an instance can be released. The workload is generated by JMeter 2.5.1 with Markov4JMeter on the SLAstic node. The JMeter profile fetches the destination IPs from the load balancer servlet at the start of a new web application call. The load balancer servlet manages a list that tracks virtual machine instances of the Tomcat image and passes a random IP to JMeter. SLAstic updates the server list by itself.

For the SLAstic node, we utilized an Amazon EC2 *m2.2xlarge* instance. This instance type provides 34.2 GB RAM and 13 EC2 Compute Units (4 virtual cores with 3.25 EC2 Compute Units each).

[3]http://sf.net/projects/ibatisjpetstore/, last accessed 2012-07-05

[4]http://www.kieker-monitoring.net/, last accessed 2012-07-05

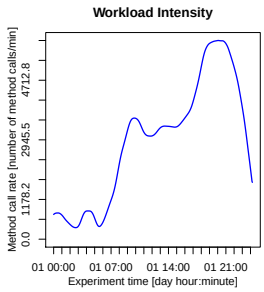


Figure 2. The used day-night-cycle workload intensity

Cost Model: We started all virtual machine instances in the region EU-West-1. Hence, we utilize the corresponding cost model. At the time of writing, an *m1.small* instance costs 0.095$ per started hour and a *c1.medium* instance costs 0.19$ per started hour.

3) Eucalyptus: The private cloud software Eucalyptus, which is deployed on our own server, is part of the evaluations. This way, we can control the overall workload intensity of the underlying hosts, which in contrast is not possible on Amazon EC2.

Our Eucalyptus server features two AMD Opteron 2384 processors that provide 8 CPU cores in sum. The server features 24 GB DDR2-667 RAM and a 1 Gigabit/s network connection. The instance type *m1.small* is configured to use a single CPU core and 1 GB RAM. The deployment is analogous to the one in Amazon EC2 and we use the same cost model which we utilize for Amazon EC2. Furthermore, we align VM instance type capacities to Amazon EC2.

4) Workload Profile: Figure 2 presents the workload intensity function that is used in the evaluations E2 and E3. The workload intensity function origins from a service provider for digital photos. It conforms to a typical day-night-cycle workload intensity often found on regional websites. In the morning the workload intensity increases until there is a first peak at noon and a second higher peak in the evening. Then, the workload intensity decreases until there are only few requests at night.

C. E1: MIPIPS Benchmark Evaluation

1) Comparisons:

Comparison MIPIPS.1: Eucalyptus m1.small instance: MIPIPS.1 compares the MIPIPS values of an Eucalyptus *m1.small* instance for 5 runs. For each run, a new instance is started and only one instance is running on our Eucalyptus setup at a time.

Comparison MIPIPS.2: Different Amazon EC2 instance types: In this comparison, we compare the MIPIPS values of different Amazon EC2 instance types. Each benchmark is

Amazon EC2 instance type	MIPIPS	EC2 compute units per core
m1.small	20.65	1
m1.large	142.13	2
c1.medium	148.81	2.5
m2.xlarge	235.57	3.25

Table I
RESULTS FOR MIPIPS EVALUATION WITH AMAZON EC2

started only once for each instance type. The instances are started in parallel on Amazon EC2.

Comparison MIPIPS.3: Amazon EC2 c1.medium instance: MIPIPS.3 analyzes the MIPIPS values of an Amazon EC2 *c1.medium* instance for 5 runs. For each new benchmark run a new instance is started. The first run is taken from MIPIPS.2 and the other four instances are started in parallel on Amazon EC2.

2) Results:

Comparison MIPIPS.1: The mean MIPIPS value of the five runs is 217.34 MIPIPS. The standard deviation amounts to 0.62 MIPIPS. The highest absolute deviation from the mean is 0.95 MIPIPS, which is about 0.44 %.

Comparison MIPIPS.2: Table I displays the results for MIPIPS.2. The *m1.small* instance is assigned with 20.65 MIPIPS. 142.13 MIPIPS are measured for the *m1.large* instance. After the *m1.large* instance, the *c1.medium* instance has 148.81 MIPIPS. At last, the *m2.xlarge* instance has the highest MIPIPS value with 235.57 MIPIPS.

Comparison MIPIPS.3: The mean MIPIPS value amounts to 148.86 MIPIPS. The standard deviation equals 0.72 MIPIPS. The highest absolute deviation from the mean is 1.18 MIPIPS, which is about 0.8 %.

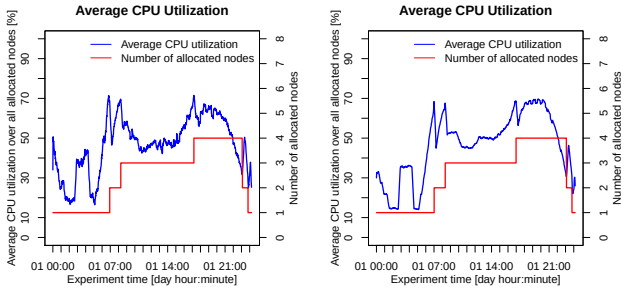
3) Discussion of the Results:

Comparison MIPIPS.1: All MIPIPS values differ by at most 0.44 % from the mean value. This value is below our 2.5 % threshold and hence, the calculated MIPIPS value is approximately constant for an *m1.small* instance on Eucalyptus.

Comparison MIPIPS.2: From *m1.small* to *m2.xlarge*, the MIPIPS values increase and the EC2 Compute Units, that can be seen as a form of performance indicator, also increase. Thus, the MIPIPS benchmark produces plausible results for those instance types.

Comparison MIPIPS.3: The MIPIPS values deviate by at most 0.8 % from the MIPIPS mean value and 0.8 % lies below our 2.5 % threshold. Therefore, the calculated MIPIPS value stays approximately constant for an *c1.medium* instance on Amazon EC2.

4) Threats to Validity: Performing only one run in comparison MIPIPS.2 might have produced MIPIPS values that largely differ from the mean value, which would result by performing more runs. However, most MIPIPS results differ by more than 10 %, and the deviation for instances that negligibly depend on the workload intensity of other instances



(a) Measured CPU utilization (b) Simulated CPU utilization

Figure 3. Average CPU utilization of allocated nodes in SingleCore.1

run on the same host was lower than 1 %. The *m1.large* and *c1.medium* instance types differ by 4 % in MIPIPS.2. Here, the measured value for *m1.large* might result from the circumstance that EC2 compute units include other factors like L2 cache and the modernity of the processor chip set. In future work, more runs and statistical methods should be conducted to evaluate the size and shape of the distribution of results.

On Amazon EC2, the performance of the instances can differ from the location where the virtual machine instances are spawned and how large the workload intensity on the running host is. For instance, this circumstance is described in [2]. Furthermore, the workload intensity on the node might have changed during the run. We cannot control these factors and thus, they stay as a threat to validity.

D. E2: Accuracy Evaluation for Single Core Instances

1) Scenarios:

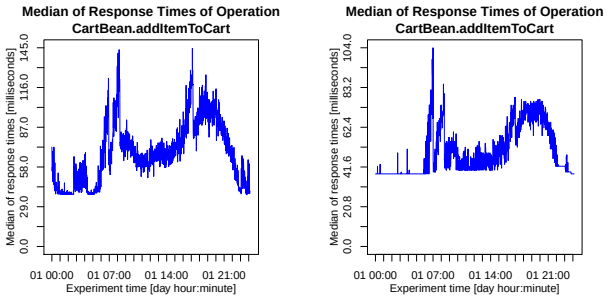
Scenario SingleCore.1: Dynamic approach for Eucalyptus run with m1.small: The workload from a run, that is conducted on Eucalyptus with *m1.small*, is used and on the basis of it, the simulation takes place. The simulation is configured to use the *dynamic approach*.

Scenario SingleCore.2: Dynamic approach for Amazon EC2 run with m1.small: The simulation takes place on the basis of the workload from a run, that is conducted on Amazon EC2 with *m1.small*. Again, the simulation is configured to use the *dynamic approach*.

2) Results:

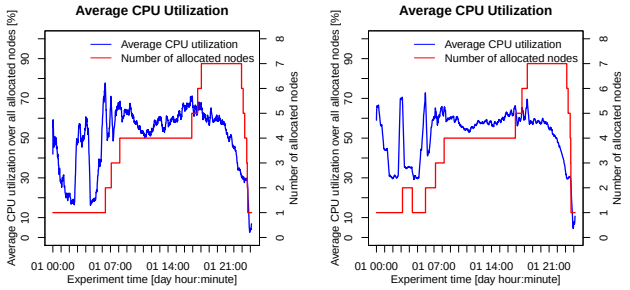
Scenario SingleCore.1: Figure 3 displays the average CPU utilization of the allocated nodes and the instance count for SingleCore.1 utilizing the *dynamic approach*. The first peak at the beginning has a lower CPU utilization in the simulated run. The rest of the experiment time, the simulation and the conducted run are roughly equal or only deviate by below 5 % CPU utilization. The instance count is approximately the same between the conducted run and the simulation.

The relative error for the CPU utilization is $RE_{CPU} = 29.18\ \%$. The relative error of the instance count is $RE_{IC} =$



(a) Measured response times (b) Simulated response times

Figure 4. Median of response times in SingleCore.1



(a) Measured CPU utilization (b) Simulated CPU utilization

Figure 5. Average CPU utilization of allocated nodes in SingleCore.2

0.64 %. The incurred costs account for 5.985$ for the Eucalyptus run. The simulation costs result in 6.365$, which is $RE_{Costs} = 6.34$ %.

In Figure 4, the median of response times per minute for SingleCore.1 are shown. The first peak in the response times of the conducted run at the beginning is smaller and is not as long as in the simulation. The peak in hour 8 is smaller in the simulation by about 10 milliseconds. The peak in hour 9 is also smaller in the simulation by about 60 milliseconds. After this peak, the simulated response times approximately follow the response times in the conducted run but they differ by an offset of about 20 milliseconds. An exception is the peak in hour 17. Here, the simulated response times differ by 70 milliseconds.

The relative error for the response times is $RE_{RT} = 24.85$ %. The overall relative error for this scenario amounts to $OverallRE = 15.25$ %.

Scenario SingleCore.2: Figure 5 displays the average CPU utilization of allocated nodes and the instance count for SingleCore.2 utilizing the *dynamic approach*. The peak in the CPU utilization at the beginning is 10 % larger in the simulation than in the conducted run. Afterwards, it drops to 20 % in the conducted run and to 30 % in the simulation.

In the simulation, there is a peak at hour 3 with 70 % CPU utilization. The instance count in the simulation increases to 2 in this hour. In contrast, the conducted run reaches 50 to 60 % CPU utilization from hour 3 to 5 and the instance count stays at 1 instance. From hour 4 to 5, the simulation has about 35 % CPU utilization. The instance count in the simulation decreases to 1 in hour 5.

The rest of the experiment time the CPU utilization and instance count is approximately equal in the simulation and conducted run.

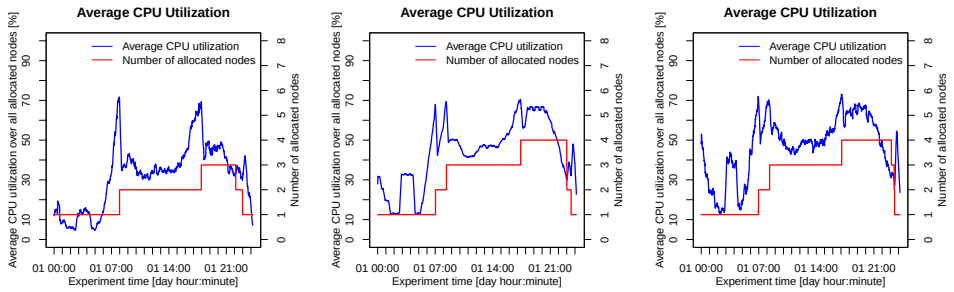
The relative error for the CPU utilization is $RE_{CPU} = 30.86$ %. The relative error for the instance count amounts to $RE_{IC} = 7.89$ %. The costs for the scenario are 8.93$ and the simulated run costs are 9.785$, which is a relative error of $RE_{Costs} = 9.57$ %. The relative error for the response times is $RE_{RT} = 42.71$ %. The overall relative error for this scenario amounts to $OverallRE = 22.75$ %.

3) Discussion of the Results:

Scenario SingleCore.1: The relative error of the CPU utilization is 29.18 % which is relatively high for the nearly equal looking CPU utilization curve. We attribute this high value to the differences when the CPU utilization is low, i.e. about 16 % in the simulation in contrast to 20 % in the conducted run. The low instance count relative error shows that the reproduction of the number of used instances of the conducted run is good and is nearly equal to it. The relative error for the costs is a bit higher than expected from the instance count relative error. The costs for the conducted run and the simulation differ by 4 paid instance hours. These difference probably occurred because the instances at the end were terminated a few minutes too late. The relative error of about 25 % for the response times is good because we do not simulate initializations of classes. These initializations typically increase the response times when a new virtual machine is started. This is the reason for the third peak to be not visible in the simulated response times. A value of 15.25 % in the overall relative error is below our threshold of 30 % and hence the simulation provides a sufficiently well reproduction of the conducted run.

Scenario SingleCore.2: The relative error of 30.86 % for the CPU utilization mainly results from the time interval between the beginning and hour 7. Here, the CPU utilization in the simulation is larger than in the conducted run. Furthermore, the second instance, which is only started in the simulation, reduces the overall CPU utilization. The second instance from hour 3 to 5 also caused the relative error for the instance count and the costs to be higher than what could have been expected of the nearly equal looking instance counts. From the high relative error for the response times we can see that the response times do not sufficiently reproduce the response times of the conducted run. From hour 7 to hour 22, the response times in the conducted run are about twice as large as in the simulation. However, since the overall relative error is 22.75 %, the scenario altogether reproduces the conducted run sufficiently.

4) Threats to Validity: For Amazon EC2, the threats we described under E1 also hold for this evaluation. An



(a) Measured CPU utilization in Amazon EC2 run

(b) Simulated CPU utilization

(c) Measured CPU utilization in Eucalyptus run

Figure 6. Average CPU utilization of allocated nodes in PredictionAmazon.1

evaluation with one program is not necessarily generalizable. With JPetStore, our simulation performs well. However, with other applications this is not necessarily the fact and should be further researched.

E. E3: Inter-Cloud Accuracy Evaluation

1) Scenario PredictionAmazon.1: Simulate with dynamic approach an Eucalyptus run from a real Amazon EC2 run: The workload is recorded with *c1.medium* instances on Amazon EC2. Then, on the basis of this workload, the simulation predicts the CPU utilization, instance count, costs, and response times for the case that the run would be conducted with Eucalyptus and *m1.small* instances. Afterwards, a run with the approximately same workload intensity is done on Eucalyptus. The simulation uses the *dynamic approach*.

2) Results: The CPU utilization and instance count for PredictionAmazon.1 are displayed in Figure 6. The left subfigure shows the conducted run on Amazon EC2. We describe the predicted CPU utilization in comparison to the afterwards conducted run on Eucalyptus. The CPU utilization of the predicted run and the Eucalyptus run are approximately the same. However, they differ at the beginning and from hour 3 to hour 5. At the beginning, the simulated CPU utilization is 30 % but the run has 52 %. From hour 3 to 5, the CPU utilization of the simulation is 32 % while the conducted run has about 40 % in this interval. The instance count is also approximately the same for both except in hour 23 where the Eucalyptus run terminates the third instance 10 minutes later than the simulation.

The relative error for the CPU utilization is $RE_{CPU} = 21.60\ \%$. The relative error of the instance count is $RE_{IC} = 1.32\ \%$. The incurred costs account for 6.175$ for the Eucalyptus run. The simulation costs result in 6.27$, which is $RE_{Costs} = 1.53\ \%$. $RE_{RT} = 38.62\ \%$ is the relative error for the response times. The overall relative error results in $OverallRE = 15.76\ \%$.

3) Discussion of the Results: The relative error for the CPU utilization is 21.60 % and thus the simulation sufficiently well predicts the CPU utilization. The relative error of 1.32 % for the instance count shows that the prediction of the number of used instances of the Eucalyptus run is nearly equal to it. The same applies for the relative error of 1.53 % for the costs. The response times relative error of 38.62 % is rather high in comparison to the other low values. We attribute this circumstance mainly to not modeling the initialization time of Java classes. The overall relative error of 15.76 % is below our 30 % threshold and thus the simulation sufficiently predicts the run using Eucalyptus.

4) Threats to Validity: The threats to validity from E2 also are applicable in this evaluation. Furthermore, we only conducted a prediction from a run with *c1.medium* instances on Amazon EC2 to *m1.small* instances on Eucalyptus.

VI. Related Work

Like CloudSim [5], GroudSim [11] is a tool for simulating clouds environments. In contrast to CloudSim, GroudSim also provides support for the simulation of Grids. Furthermore, GroudSim utilizes an event-based simulator that requires only one thread per simulation, while CloudSim follows a process-based approach that runs a separate thread for each entity. The equivalent to Cloudlets in CloudSim are GroudJobs in GroudSim. A further feature of GroudSim is the definition of failures. Failures can be generated in a defined interval for a specific registered resource.

SLAstic.SIM [10] is a performance simulator for runtime configurable component-based software systems utilizing SLAstic. The system, that should be simulated, must be modeled as an instance of the Palladio Component Model (PCM) for SLAstic.SIM. Furthermore, SLAstic.SIM requires external workload traces and reconfiguration plans for simulation. SLAstic.SIM helps to predict the performance impact of specific reconfiguration actions and thus it can support the evaluation of different adaptation strategies.

iCanCloud [12] is a simulation platform for modeling and simulating cloud computing architectures. It is mainly aimed for the prediction of the trade-off between costs and performance of a specific application in a specific cloud environment and configuration. Furthermore, it bases on the

SIMCAN simulation framework. With iCanCloud, the user can model applications using traces of real applications, using state graphs, and by programming new applications directly in the simulation platform. However, it does not provide support for importing existing software systems easily. These must be modeled manually.

The Cloudstone toolkit [13] includes a multi-platform, multi-language benchmark, and measurement tools for Web 2.0. It has three components. The first component comprises of *Olio* and *Faban*. *Olio* consists of two implementations of a social-event calendar web application. Both implementations feature user-generated content, social networking functions, and an AJAX-based user interface. *Faban* is an open source performance workload creation and execution framework. The second component is a set of automation tools for, e.g., database population and metric gathering for testing *Olio*. A recommended method for calculation of the metric *dollars per user per month* forms the third component.

SMICloud [14] is a framework for comparing different cloud providers based on the requirements of the user. It utilizes different measures that form a Service Measurement Index (SMI) for each cloud provider. With SMICloud a user can compare between different cloud provider offerings basing on his priorities and requirements.

VII. Conclusion and Future Work

During a cloud migration a cloud user has to assess a wide range of different cloud deployment options (CDOs). For example, a selection of a cloud provider must be conducted. Furthermore, the mapping between services and virtual machine instances must be considered. The virtual machine instances' configuration and adaptation strategies must be also specified. Rewriting and testing the software with the different cloud deployment options is infeasible. Simulating the different deployment options can assist to find the best ratio between high performance and low costs.

The article showed how CDOs can be simulated. First, the diverse inputs and outputs of the simulation were described. An approach for the derivation of MIPIPS, a new measure for the computing performance of nodes, and three approaches for instruction count (IC) derivation were presented. The MIPIPS and weights benchmark and CDOSim were described. The evaluation showed that CDOSim's simulation results are reasonable near to the conducted runs concerning accruing costs and performance. Especially, we demonstrated that CDOSim can sufficiently accurate predict the execution on a different cloud provider.

Most future work lies in further adaptations to CDOSim. In order to perform an automatic optimization of cloud deployment options efficiently, it should be possible that simulations can run in parallel. Hence, CloudSim should be extended to support parallel simulations. Furthermore, it should be possible to simulate other attributes of the cloud like the I/O performance. In addition, the elementary model for computing network costs should be extended.

References

[1] J. Grundy, G. Kaefer, J. Keong, and A. Liu, "Guest Editors' Introduction: Software Engineering for the Cloud," *IEEE Software*, vol. 29, pp. 26–29, 2012.

[2] P. Brebner and A. Liu, "Performance and Cost Assessment of Cloud Services," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, E. Maximilien, G. Rossi, S.-T. Yuan, H. Ludwig, and M. Fantinato, Eds. Springer Berlin/ Heidelberg, 2011, vol. 6568, pp. 39–50.

[3] F. Fittkau, "Simulating Cloud Deployment Options for Software Migration Support," Master's thesis, Software Engineering Group, University of Kiel, Kiel, Germany, March 2012.

[4] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems," *Computer Standards and Interfaces*, vol. 33, no. 6, pp. 519–532, 2011.

[5] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, pp. 23–50, Jan. 2011.

[6] S. Frey, W. Hasselbring, and B. Schnoor, "Automatic Conformance Checking for Migrating Software Systems to Cloud Infrastructures and Platforms," *Journal of Software Maintenance and Evolution: Research and Practice*, doi: 10.1002/smr.582, 2012.

[7] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, pp. 57–76, Oct. 2007.

[8] B. P. Zeigler, *Theory of Modelling and Simulation*. Malabar: Krieger, 1985.

[9] D. A. Menasce and V. A. F. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall International, Sep. 2001.

[10] R. von Massow, A. van Hoorn, and W. Hasselbring, "Performance simulation of runtime reconfigurable component-based software architectures," in *Software Architecture*, ser. Lecture Notes in Computer Science, I. Crnkovic, V. Gruhn, and M. Book, Eds. Springer Berlin / Heidelberg, 2011, vol. 6903, pp. 43–58.

[11] S. Ostermann, K. Plankensteiner, R. Prodan, and T. Fahringer, "GroudSim: An Event-based Simulation Framework for Computational Grids and Clouds," in *CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing*. Springer, 2010.

[12] A. Nuñez, J. Vázquez-Poletti, A. Caminero, J. Carretero, and I. Llorente, "Design of a new cloud computing simulation platform," in *Computational Science and Its Applications - ICCSA 2011*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, vol. 6784, pp. 582–593.

[13] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proceedings of the 1st Workshop on Cloud Computing (CCA 08)*, Oct. 2008.

[14] S. Garg, S. Versteeg, and R. Buyya, "SMICloud: A Framework for Comparing and Ranking Cloud Services," in *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC 11)*, Dec. 2011, pp. 210–218.