A Framework for Automated Software Testing on the Cloud

Gustavo Sávio de Oliveira, Alexandre Duarte
Informatics Centre
Federal University of Paraiba
Joao Pessoa, Paraiba, Brazil
gso@di.ufpb.br, alexandre@ci.ufpb.br

***Abstract*— This work presents the framework CloudTesting, a solution to parallelize the execution of a test suite over a distributed cloud infrastructure. The use of a cloud as runtime environment for automated software testing provides a more efficient and effective solution when compared to traditional methods regarding the exploration of diversity and heterogeneity for testing coverage. The objective of this work is evaluate our solution regarding the performance gains achieved with the use of the framework showing that it is possible to improve the software testing process with very little configuration overhead and low costs.**

***Keywords*— software testing, cloud**

I. Introduction

An effective software testing process must be performed quickly and automatically. There are well-established solutions [2] to automate the software testing process and other solutions aimed mainly at speeding up the process by distributing the execution of a test suit over a set of processors [4] [3]. In the same context, there are also efforts to explore the characteristics of distributed computing platforms such as grids, its extensive parallelism and vast heterogeneity of environments, in order to limit the effects of the development environment on the results of tests [5]. However, only recently new studies have adopted cloud computing as a platform for testing software in large-scale [6] [7].

Exploring cloud computing platforms for running software tests can provide significant gains in both efficiency and effectiveness of testing when compared to traditional methods [15]. Several factors contribute to this statement, such as reducing deployment, maintenance and licensing environments costs, flexibility for acquisition and installation of customized test environments on demand and scalability in a timely and economical way [14].

However, development and testing process in the cloud usually require significant efforts in the configuration, distribution, and execution of tests [6]. In order to facilitate the exploration of cloud computing platforms and environments for software testing we propose a framework called CloudTesting. Our solution enables parallel execution of automated software tests in heterogeneous environments, decreasing the time spent during test through an abstraction layer for users, eliminating the need to perform complex configurations. A very important feature of the proposed solution that may facilitate its adoption is that the tool does not require any source code modification to execute the software tests in the cloud.

We evaluated our solution by conducting some experiments using resources provided by Amazon EC2 in comparison with the execution of the same tests performed locally. The quantitative analysis performed indicates significant gains in execution time using the cloud infrastructure with very low configuration overhead and additional cost.

II. The CloudTesting FrameWork

Large software projects usually include a proportionately large amount of test cases [5]. Often, these tests are time consuming to run, hardening the use agile development process that heavily rely on automatic testing, such as *Extreme* Programming [8]. Since each test consumes a certain period of execution, depending on the size and complexity of the software the only way to decrease the time spent in the testing process is a massive parallelization of their execution [9].

The *CloudTesting*[1] Framework improves this process by encapsulating from the developer/tester all the complexity involved in the parallel execution of test cases using on-demand computational resources without requiring any modification to the source code of tests for its use.

When it is decided to distribute and parallelize the tests, the result is a significant reduction in the time required to execute a large test set, reducing the time spent in identifying and correcting faults, which represents a large impact on the total cost of development. Furthermore, the framework increases the reliability of the test results by using heterogeneous and uncontaminated environments for running the tests, facilitating the exposure of failures that would otherwise be hidden until the software production stage.

The framework is initially focused on software developed in Java. It distributes the execution of a set of unit tests by performing a reflection on the local classes that contains the tests and scheduling the execution of each test on a different machine on the cloud. The load over the available machines is balanced using an implementation of the Round-Robin algorithm [10]. Thus, all requests are evenly distributed among the machines participating in the test infrastructure.

[1] Available in: http://code.google.com/p/cloudtestingdi/

Fig. 1 presents the architectural components of the *CloudTesting* framework, which is composed by the *configuration, reflection, distribution, connection, log* and *main* components.

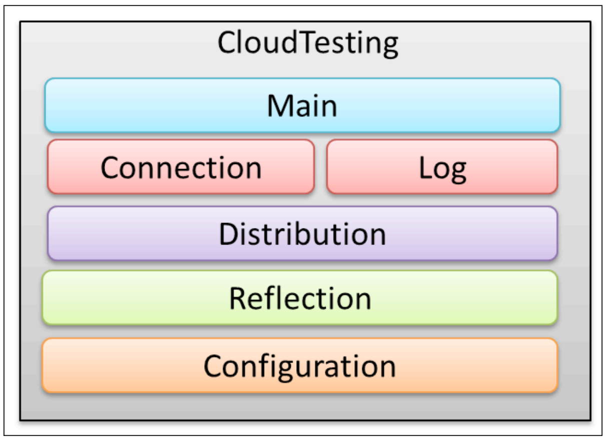

Fig. 1. *CloudTesting* components

The *configuration* component assists in definition of information for *paths, hosts* and *load balancing.* It is used, for example, to specify the local storage space for test results, the libraries that should be transferred to the cloud for the proper test execution and the file access permission on the cloud provider. It encapsulates also the list of machines available for test execution on a given moment and the parameters for the load balancer. The *Reflection* component is responsible for extracting the tests cases in order to inform the *distribution* component what test methods should be executed in the cloud.

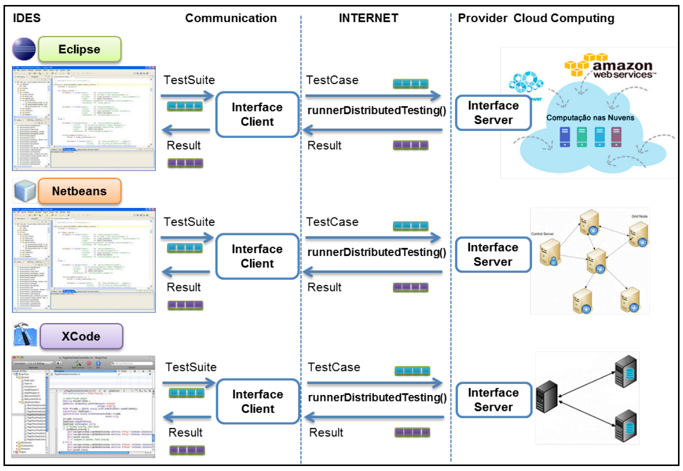

Fig. 2. *CloudTesting* intermediating the distributed execution of automated tests on different parallel infrastructures

Fig. 2 depicts the *distribution* component being used to intermediate the execution of test suites over a parallel infrastructure. To work with a given IDE and parallel infrastructure the framework must be extended to include specific plugins. The current implementation provides plugins for the Eclipse IDE and the Amazon Web Services infrastructure.

The *connection* component provides an interface on the client side to communicate with the cloud provider. In the cloud site this component provides a service that manages the execution of each test and sends the test results back to the client in real time. The *log* component records events generated in the process. The *main* component is a facade to encapsulate all the previous components.

III. Related Work

It's observed that during the last years several well-founded studies that approach solutions and means to automate and accelerate the software testing process have been developed [2] [4] [3]. Nevertheless, as the computer systems grow larger and more complex, software testing requires more effort. Various automatic distributed software testing systems or large scale systems have been proposed in recent years, since this methodology exploits the characteristics of wide parallelism and extensive heterogeneity of environments in order to limit the effects of the development environment on the test results [17]. Some studies that relate to our solution are discussed below.

The GridUnit tool [17] investigates the use of computational grids as a test environment and presents an open source solution for running unit tests automatically in the grid. The solution is an extension of the JUnit framework [2], able to distribute the execution of a JUnit test suite in the grid without the need for any modification in the source code. The GridUnit have five main characteristics. Transparent and automatic distribution: each JUnit test is regarded as an independent task and the scheduling of executing the task in the grid is done without human intervention; Test for prevention of contamination: each test is run using virtualization offered by the grid, preventing that the A1 test results change the A2 test results; Test load distribution: the tool has the Grid Scheduler that manages load distribution; Test suite integrity: as previously mentioned, each JUnit test runs as an independent task. For each new test, GridUnit creates a new instance of the Test class, makes calls to the setUp (), testMethod (), tearDown () methods and then removes the instance; Test execution control: The GridTestRunner and GridTestListener interfaces provide means for establishing the execution and monitoring of tests in a centralized manner.

Our solution (CloudTesting) differs from GridUnit regarding to the infrastructure and abstraction of complexities. GridUnit uses computational grids and CloudTesting supports the use of different distributed execution platforms to perform automatic testing of an application using a variety of runtime environments, such as the cloud. The advantage of using the cloud is that it provides automatic resizing of virtualized hardware resources, security through virtualization, no worry about workflow, easy management, usability and flexible business model.

The D-Cloud work [9] proposes a solution for testing parallel or large scale distributed systems that require characteristics of highly reliable systems, focusing on fault tolerance test at the hardware level. The research introduces the infrastructure of cloud computing for software testing, being composed of multiple nodes of virtual machine that run operating systems hosted with fault injection, a controller node manages all hosted operating systems, and a frontend which controls hardware and software configurations and test scenarios. The conceptual architecture of D-Cloud highlights the following properties:

- Virtual machine with Fault Injection - the FaultVM which is based on QEMU as the virtualization software;
- Management of computational resources using Eucalyptus - to manage the large amount of resources the Eucalyptus software is used to perform the management of the cloud. Thus relieving the responsibility of the tester to manage the allocation of computing resources, this process is done automatically;
- System automated testing and configuration - the tool automates the system configuration and the testing process, including the fault injection, based on scenarios described by a tester;
- Preparation of the test scenarios - it is established through a file written in XML, while providing multiple scenario files, several systems can be tested in parallel.

The D-Cloud work has different context and approach to the execution of automated software testing when confronted with the proposal of CloudTesting. Basically, the direction of automated software testing diverges. The first project directs its tests for fault tolerance at the hardware level, the second research points to the execution of a set of unit tests making use of the JUnit framework. In addition, D-Cloud was designed to work exclusively for the infrastructure of cloud computing, ignoring other platforms and modes of execution. Since it is a framework, CloudTesting can be adapted to specific needs.

IV. Experimental Results

The CloudTesting framework can be extended to be used in various integrated development environments (IDE) and to obtain resources from diverse execution platforms. However, for this analysis, we developed an instantiation of the framework for the Eclipse IDE and the cloud provider Amazon AWS.

To perform the experiments, we developed a set of 1800 tests that have an average processing time previously known when executed on a local machine, aiming to make a comparison with the results obtained through the use of the framework.

The experiments are divided into two scenarios: (01) where the test suite is executed 45 times on a local machine, (02) where the test set is distributed 45 times with resources obtained from the cloud provider.

During the analysis we used the Chauvenet's criterion [13] to remove outliers. Then calculate the average execution time, the standard deviation and identified the best and worst execution times. This information allows the calculation of the speedup (SP = T1/Tp where T1 is the execution time of the sequential program and Tp is the execution time of the same program running in parallel) and the efficiency of the parallel execution (EF = Sp / Np where Sp is the speedup achieved and Np is the number of cloud machines used to run the tests in parallel). After this we calculated the 95% and 99% confidence intervals [12].

A. Scenario 01

The tests executed locally followed a strict policy regarding the use of the equipment during the test period. Aiming to avoid anomalous results and get real results, we used a dedicated machine to run the tests. After the end of each test we *rebooted* the machine in order to sanitize data stored in the RAM and processor *cache.* The machine setup had Intel Core 2 Duo 2.20 GHz, 4 Gigabytes of RAM and a 32-bit Linux Operating System.

To capture the actual runtime of the test we used the integrated development environment Eclipse with the JUnit plugin PDE that has a default profiler and manages all unit tests software executed with the *JUnit* platform.

In this scenario, each unit test achieved an average runtime close to 1 second. The set comprised 1800 test, so a single run of the test set took, on average, 30 minutes to complete. The total running time for the 45 rounds was 22:54:51. The average execution time was 0:30:33 with a standard deviation of 1.47% (0:00:27), best and worst execution time were 0:29:58 and 0:31:09. With a 95% confidence interval the lower limit and upper limit were 00:30:25 to 00:30:41 and with a 99% confidence interval the lower and upper limit were 00:30:23 and 00:30:43. These data provide a basis for analyzing the *speedup* of using the *CloudTesting* framework.

B. Scenario 02

To evaluate the execution time of the tests on the cloud we have to consider factors like network latency and fluctuation. Therefore, we established a common time scale for the tests in order to obtain theoretically the same bandwidth conditions for all experiments. All tests were performed in a time range between 00:00 and 04:00, a time slice selected to reflect the low network usage in the laboratory. Due to the amount of testing, the experiments were not performed in a single day, instead, were performed on alternate days following the policy described below.

The 45 rounds were repeated three times using different Amazon instance types: Micro, Small and Medium. We used 18 instances of each type to conduct the three sub-sets of experiments. To capture the actual runtime of the test we used the Eclipse integrated development environment with TPTP plugin profiler. We used a network with a nominal bandwidth of 15Mbps for download and 1mbps for upload. Some configuration settings were applied to the cloud machines prior to the execution of the tests: (1) The log and lib directories were created for storing logs and libraries respectively, (2) the JUnit and CloudTesting libraries were distributed to the lib directory, (3) The CloudTesting remote service was enabled.

1) Scenario 02 - Micro Instances

Micro instances have limited CPU resources, while allowing the increased computing capacity when additional cycles are available. They are composed of 32-bit Linux Operating System with up to 2 EC2 compute units (for periodic small bursts), 613MB memory RAM only EBS storage performance and low input and output data [11].

For this scenario we obtained a total execution time of 08:04:53, an improvement of 14:49:58 over the total execution time from Scenario 01. The average execution time was 0:10:46 with standard deviation of 18.4% (0:01:59), the best and worst running time were 0:03:56 and 0:12:03. Analyzing the graph with all execution times (as illustrated in Fig. 3) we can see that there is a considerable variation with a direct impact on the resulting confidence intervals. With a 95% confidence interval the lower limit and upper limit for the running time were 00:10:12 and 00:11:21 and with a 99% confidence interval these limits were 00:10:01 and 00:11:32 respectively.

Regarding the wide variation in the execution time of some experiments, we used the Chauvenet criterion to check whether they could be considered as anomalies. However, no results fell within this situation. This leads to or conclusion that this difference was due to the condition that the *micro instances* are able to surpass for a short time the 2 EC2 compute units. Another hypothesis relates to network resources and shared disk subsystem, possibly when a resource is underutilized.

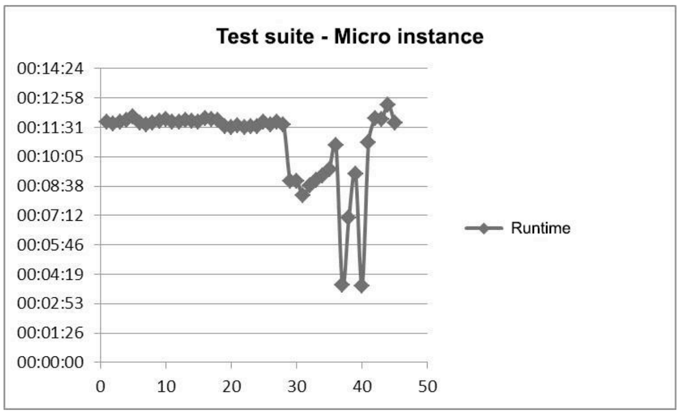

Fig. 3. *Execution time for the Scenario 2 – Micro instances*

2) Scenario 02 – Small Instances

Instances of type Small are part of the group of standard instances recommended for the execution of most applications. For our experiment all instances had Linux 32-bit distribution, 1 EC2 compute unit, 1.7 GB of RAM and 160 GB available for data storage.

The total execution time for the test set was 4:07:31, an improvement of 18:47:20 from scenario 01 and of 3:57:22 when compared to the use of Micro instances. The average execution time was 0:05:29, with a standard deviation of 2,3% (0:00:08), the best and the worst run times were 0:05:19 and 0:05:52 respectively. The runtime for each individual experiment was consistent as can be concluded with the narrow standard deviation. The 95% confidence interval showed a lower and upper limit of 00:05:26 and 00:05:32 and in the 99% confidence interval the limits were 00:05:21 and 00:05:37.

3) Scenario 02 – Medium Instances

Instances of the type *Medium* are also part of the standard group and have the same purpose of the instances of type *Small*. During the experiments we used the following configuration: high performance CPU, 3.75 GB of memory, two processing units EC1 (1 virtual core with 2 EC2 Processing Units each), 410 GB of data storage and a 32-bit Linux distribution.

The total execution time of the test set was 2:50:48, a reduction of 20:04:03 compared to Scenario 01, 05:14:05 when compared to running on *Micro instances* and 01:16:43 compared to running on *Small* instances.

The average execution time for a test was 0:03:47, a standard deviation of 3,9% (0:00:09), the best and worst execution times were 00:03:12 and 00:04:01 respectively. The 95% confidence interval showed a lower and upper limit of 00:03:44 and 00:03:50, and the 99% confidence interval showed a lower and upper limit 00:03:43 and 00:03:51.

C. Discussion

Some points should be considered before evaluating a particular *speedup* in positive or negative way. Ideally, since we used 18 machines do execute the tests one would expect to be 18 times faster than using a single machine. However, we must remember that in order to parallelize the execution using cloud machines we need to upload the code that will be executed remotely. So, depending on the size of the project, distributing packets over the network can be costly. Another factor is directly related to the processing power of the machines and the ability to input and output data in the cloud. Finally, there is also the relationship with the virtualization server since the performance will often depend on the amount of resources available on physical server.

We observed in the experiments that the *micro* instances achieved a considerable speedup, despite not being suitable for large load requests in a short period of time. For this scenario the *medium* instances are more adequate.

The experiments performed using the *micro* instances showed a *speedup* of 8.55x and a 0.48 parallel efficiency for the best runtime and a speedup of 2.61x and efficiency of 0.14 in the worst case. On average, it showed a speedup of 2.89 what results in a parallel efficiency of a 0.16.

Tests executed using instances of type *small* were surprisingly regular, showed a *speedup* of 5.71x and an efficiency of 0.32 on the best case and 5.70 e 0.31 on the worst case. On average the results are approximately the same as in the best case.

The experiments performed using the *medium* instances reached a *speedup* of 9.48x and a 0.53 parallel efficiency on the best case and a speedup of 8.72 and a 0.48 parallel efficiency on the worst case. On average we obtained a speedup of 7.83 and a 0.43 parallel efficiency. These results are summarized on Fig 4.

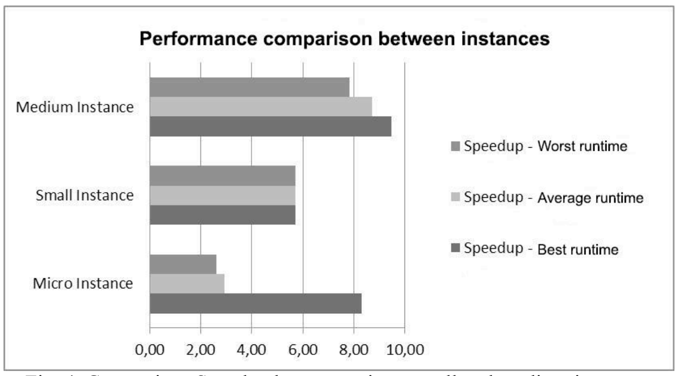

Fig. 4. Comparison Speedup between micro, small and medium instances

One curious observation was that in some runs we obtained higher speedups with the cheaper *micro* instances than with the *small* instances.

This result was surprising at first, though, as previously mentioned, the *micro* instances can momentarily use up to 2 ECUs, getting twice the computing power of a *small* instance. Nevertheless, on average, *micro* instances are quite slower than the *small* instances.

As expected, the best results were obtained with the higher priced *medium* instances. This result was expected due to hardware configuration and the better input and output rates over the other tested instances.

The costs involved with the execution of 45 rounds of experiments were US$ 16.20 for the 18 micro instances, US$ 68.85 for 18 small instances US$ 137.70 for the 18 medium instances.

Therefore, we had a cost of US$ 0.36 per round using *micro* instances, US$ 1.53 per round using *small* instances and US$ 3.06 per round using *medium* instances.

Considering the costs and the speedup we can calculate a cost/benefit ratio by dividing the round cost by the average speedup for each type of instance, achieving the following results:

- Micro instance: US$ 0.36 / 2.61x = 0.138
- Small instance: US$ 1.53 / 5.70x = 0.268
- Medium instance: US$ 3.06 / 7.83 = 0.391

This result shows that the cheaper instances have a better cost/benefit ratio that higher priced instance types.

V. Conclusions

The experimental results indicate significant performance gains with the distribution of the execution of software tests without significantly increasing the costs involved in assembling the infrastructure, thus facilitating the process of using cloud infrastructures as a executing platform for automated software testing.

The *CloudTesting* framework simplifies the execution of automatic tests in distributed environments, obtaining gains in performance, reliability and simplicity of configuration by running parallel automatic software tests in heterogeneous environments through an abstraction layer for users.

Acknowledgment

Authors would like to thank CNPQ (research grant MCT/CNPQ 09/2010), CAPES and Amazon for their financial support.

References

[1] Smith, A. and Jones, B. (1999). On the complexity of computing. In *Advances in Computer*Science,pages 555-566. Publishing Press.

[2] Gamma, E. and Beck, K. (1999). JUnit: A cook's tour. Java Report, 4 (5) :27-38.

[3] Hughes, D. and Greenwood, G. (2004). A Framework for Testing Distributed Systems, In Proceedings of the 4th IEEE International Conference on Peer-to-Peer computing.

[4] Kapfhammer, G., M. (2001). Automatically and transparently Distributing the Execution of Regression Test Suites. In Proceedings of the 18th International Conference on Testing Computer Software.

[5] Duarte, A. *et. al.* (2005). GRIDUNIT: software testing on the grid, Proceedings of the 28th international conference on Software engineering. New York, USA, p. 779-782.

[6] Hanawa, T. *et al.* (2010). Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems, Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference.

[7] Oriol, M. and Ullah, F. (2010). YETI on the Cloud. Software Testing, Verification, and Validation Workshops (ICSTW) Third International Conference 2010.

[8] Wu, X. and Sun, J. (2010). The Study on an Intelligent General-Purpose Automated Software Testing Suite, Intelligent Computation Technology and Automation (ICICTA) International Conference 2010.

[9] Banzai, and Takayuki Koizumi, Hitoshi (2010). D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology Cluster, Cloud and Grid Computing (CCGrid), 10th IEEE / ACM International Conference.

[10] Ramabhadran, S. and Pasquale, J. (2003). "Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay," Proc. of SIGCOMM.

[11] 'AmazonEC2 (2012). "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/pt/ec2/instance-types/,June.

[12] Dillard, GM (1997). "Confidence intervals for power Estimates," Signals, Systems & Computers, 1997. Conference Record of the Thirty-First AsilomarConference.

[13] Pop, S.; Ciascai, I. and Pitica, D. (2010), "Statistical analysis of experimental data Obtained from the optical pendulum," *Design and Technology in Electronic Packaging (SIITME), 2010 IEEE 16th International*Symposium.

[14] Grundy, J. *et al.* (2012), "Guest Editors' Introduction: Software Engineering for the Cloud, "Software, IEEE, vol.29, no.2, pp.26-29.

[15] Riungu-Kalliosaari, L.; Taipale, O. and Smolander, K. (2012). "Testing in the Cloud: Exploring the Practice," Software, IEEE, vol.29, no.2, pp.46-51.

[16] Andrade, Nazareno *et al.* (2003). OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 61-86.

[17] Duarte, A. et al.. Multi-environment Software Testing on the Grid. In: PADTAD ’06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging. New York, NY, USA: ACM, 2006. p. 61–68. ISBN 1-59593-414-6.