2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing

D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology

Takayuki Banzai†, Hitoshi Koizumi†, Ryo Kanbayashi†,
Takayuki Imada†, Toshihiro Hanawa†,‡ and Mitsuhisa Sato†,‡
† Graduate School of Systems and Information Engineering, University of Tsukuba
‡ Center for Computational Sciences, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan
{banzai,koizumi,kanbayashi,imada,hanawa,msato}@hpcs.cs.tsukuba.ac.jp

***Abstract*—In this paper, we propose a software testing environment, called D-Cloud, using cloud computing technology and virtual machines with fault injection facility. Nevertheless, the importance of high dependability in a software system has recently increased, and exhaustive testing of software systems is becoming expensive and time-consuming, and, in many cases, sufficient software testing is not possible. In particular, it is often difficult to test parallel and distributed systems in the real world after deployment, although reliable systems, such as high-availability servers, are parallel and distributed systems. D-Cloud is a cloud system which manages virtual machines with fault injection facility. D-Cloud sets up a test environment on the cloud resources using a given system configuration file and executes several tests automatically according to a given scenario. In this scenario, D-Cloud enables fault tolerance testing by causing device faults by virtual machine. We have designed the D-Cloud system using Eucalyptus software and a description language for system configuration and the scenario of fault injection written in XML. We found that the D-Cloud system, which allows a user to easily set up and test a distributed system on the cloud and effectively reduces the cost and time of testing.**

I. Introduction

In today's information society, high reliability is a very important factor in various information systems. Because of the difficulty of ensuring dependability, the growth of the scale of software and software testing, in many cases, sufficient software testing is not possible. In particular, although reliable systems such as high-availability servers are parallel and distributed systems, it is often difficult to test a parallel and distributed system in the real world after deployment. Highly dependable systems must have some mechanism for fault tolerance for not only software bugs but also for hardware failures. In order to test system with respect to correct operation, including handling of hardware failures, we must test the system under a number of hardware failures. However, it is difficult to destroy a particular component of an actual piece of hardware or to place a heavy load on a hardware device.

In addition, on a distributed system that consists of multiple physical server nodes, it is very difficult to consistently cause similar errors during a test and difficult to analyze the cause of failure. The present solution is to use virtual machine technology to provide the fault injection facility. We have designed a virtual machine based on QEMU [2], called FaultVM, which emulates hardware failures of several devices at the level of the virtual machine. FaultVM allows the user to inject faults into the guest OS on which the tested software is executed. In order to manage virtual machines, we introduce the Eucalyptus cloud computing system.

We propose a distributed system for a testing environment, referred to as D-Cloud, which provides parallel software testing environments for reliable distributed and single systems using virtual machines with fault injection. We have been implemented D-Cloud using the Eucalyptus cloud computing system [1] as a base system. The contributions of this paper are summarized as follows:

- We propose a software testing environment for distributed systems as a new application of cloud computing technology.
- Using a virtual machine with fault injection facility, the user can test functions of fault tolerance in reliable distributed system software.
- We designed the description format of system configuration and test scenario to automate the setup and test execution.

The features and objectives of the D-Cloud system are described in Section 2. The design of the D-Cloud is described in Section 3. The scenario description for testing is presented in Section 4, and D-Cloud operation is described in Section 5. Related research is described in Section 6. Finally, we conclude the present study and describe future research in Section 7.

II. D-Cloud program testing environment

D-Cloud provides a virtual machine environment that is specialized for fault injection in order to develop a highly reliable system. The features of D-Cloud are listed as follow.

- D-Cloud enables testing of fault tolerant functions with respect to hardware failures that occur in a physical

978-0-7695-4039-9/10 $26.00 © 2010 IEEE
DOI 10.1109/CCGRID.2010.72

machine using the fault injection facility implemented in the virtual machine layer.

- The computing resources can be managed flexibly. If resources are available, then test cases can be executed quickly by simultaneously using the resources.
- D-Cloud automates testing using descriptions of the system configuration and the test scenario to execute tests on cloud computing systems.

A. Fault injection in a virtual machine

D-Cloud uses a virtual machine to execute system tests. The virtualized hardware device can simulate failures on the guest OS, and fault injection using virtual machines allows system tests to be executed without changing the program. Using the virtual machine, D-Cloud can test software running in not only the userland layer but also in the kernel layer. When software bugs on the kernel layer are detected during a system test, the OS may hang-up automatically due to a kernel panic. When the system runs on real machine, it is difficult to obtain helpful information for the bug fix in this case, because the user cannot manipulate the OS under the kernel panic. However, when using a virtual machine, a bug in the OS running on the virtual machine does not affect the host OS running on a physical machine. Therefore, the tester can continue system tests, and the tester can collect information for debugging even if the guest OS crashes. Furthermore, the snapshot of the previous state in the guest OS permits the operation to return until the desired state repeatedly.

B. Management of computing resources

For developing reliable systems, it is important that system tests must be executed for as many cases as possible in order to find and fix as many bugs as possible. In addition, in order to execute many tests, a large amount of resources must be managed efficiently and flexibly. In D-Cloud, resources are managed by a cloud computing system. For example, a number of systems that require high reliability and dependability consist of multiple nodes linked by a network. In this case, D-Cloud can test such distributed systems using several guest OSes.

C. Automating system configuration and testing

D-Cloud automates the system setup process of the tested system and the test process, including the fault injection, based on a scenario written by a tester. When the tester writes a number of configurations of system test environments in a scenario description file, D-Cloud sets up appropriate test environments and executes appropriate tests automatically. Therefore, D-Cloud enables the behavior of dependability functions on the system to be tested exhaustively and enables system tests to be executed quickly.

III. Design of D-Cloud

D-Cloud uses QEMU as virtualization software and Eucalyptus, which is an open-source implementation having the same API as AmazonEC2[3]. Figure 1 shows an overview of D-Cloud. D-Cloud consists of the following components:

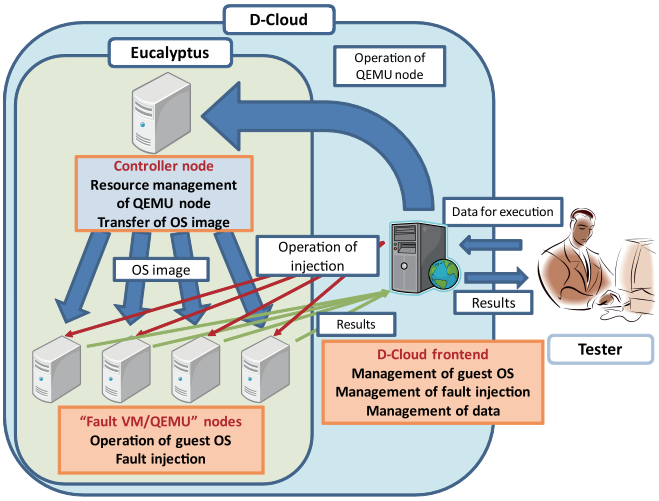

Fig. 1. Configuration of D-Cloud

1) QEMU nodes, which are available with the fault injection facility.
2) The controller node, which manages the QEMU nodes using Eucalyptus.
3) D-Cloud frontend, which issues test and fault injection commands and transfers input/output data with QEMU nodes.

A. Fault injection using QEMU

D-Cloud uses QEMU as a virtualization software. The advantages of using QEMU are described following.

- QEMU can emulate a number of hardware devices. Thus, QEMU may control several hardware faults in the guest OS.
- The QEMU source codes are available as open-source software. This allows modification to the hardware emulation codes in order to add the fault injection function.
- QEMU can isolate the host OS from the guest OS. In addition, QEMU may insulate the computing host from anomalous behavior of the guest OS during various tests.

1) Type of fault injection: Table I lists the types of fault injection . Here, we implement the fault injection mechanism for the hard disk controller, the network controller, and the memory as target devices.

2) Reproducibility in fault injection: Reproducibility is important in fault tolerant testing using fault injection. If a testing environment has reproducibility, then the tester can find errors that occur under specific conditions and can fix such bugs. In particular, in many cases of hardware device errors, we cannot predict the effects of bugs, and these effects are usually not repeatable. However, since hardware errors can be emulated anytime and anywhere using a virtual machine, the costs associated with using a virtual machine are lower than those associated with using physical machines.

TABLE I
TYPES OF FAULT INJECTION

device	contents	value
Hard disk	Error of specified sector	badblock
	Specified sector is read-only	readonly
	Error detection by ECC	ecc
	Received data contains error	corrupt
	Response of disk becomes slow	slow
Network	1bit error of packet	1bit
	2bit error of packet	2bit
	Error detection by CRC	crc
	Packet loss	loss
	NIC is not responding	nic
Memory	Bit error	bit
	Byte at specified address contains error	byte

B. Managing resources using Eucalyptus

D-Cloud manages virtual machine resources using Eucalyptus. Eucalyptus is a cloud computing infrastructure that manages machine resources flexibly using a virtual machine. D-Cloud consists of multiple QEMU nodes, which execute guest OSes, and a controller node, which controls all of the guest OSes.

The procedures to manage machine resources are shown as follow.

1) A tester uploads OS images to the controller node and registers the machine images to the D-Cloud frontend.
2) The controller node transfers OS images to QEMU nodes.
3) The OS images are booted as a guest OS on QEMU nodes.

After the controller node receives the request to boot a guest OS, the controller node transfers the OS images to QEMU nodes, which are available to run the OS images. Thus, the tester does not need to be aware of computing resources on D-Cloud.

C. D-Cloud frontend

D-Cloud frontend manages guest OSes, configures system test environments, and transfers various data from the tester to a guest OS executed for the purpose of system testing.

D-Cloud frontend performs its function as follows:

1) D-Cloud frontend receives a test scenario, a test program, input data, and an execution script from a tester.
2) D-Cloud frontend then issues a request to boot a guest OS to the controller node.
3) D-Cloud frontend then transfers the test program, the input data, and the execution script to the guest OS.
4) D-Cloud frontend then issues the fault injection command to the target guest OS.
5) Finally, D-Cloud frontend collects the output data, logs, and snapshots.

Since D-Cloud frontend collects data obtained in the test, the tester can download these data anytime. If the tester checks the output and traces the operation using saved snapshots, the tester can discover some bugs and investigate these bugs in detail.

```
<jobDescription>
  <machineDefinition>
    <machine>
      <name>nodeA</name>
      <cpu>1</cpu>
      <mem>512</mem>
      <nic>2</nic>
      <id>id-1</id>
    </machine>
  </machineDefinition>
  <systemDefinition>
    <system>
      <name>systemA</name>
      <host>
        <hostname>node0</hostname>
        <machinename>nodeA</machinename>
        <config>nodeconf</config>
      </host>
    </system>
  </systemDefinition>
  <injectionDefinition>
    <injection>
      <name>injectionA</name>
      <fault>
        <location>network</location>
        <target>eth0</target>
        <kind>loss</kind>
        <time>40</time>
      </fault>
    </injection>
  </injectionDefinition>
  <testDescription>
    <run>
      <name>testA</name>
      <systemname>systemA</systemname>
      <halt when="400">down</halt>
      <script>
        <on>node0</on>
        <putFile>files</putFile>
        <exec>script</exec>
        <inject when="200">injectionA</inject>
      </script>
    </run>
  </testDescription>
</jobDescription>
```

Fig. 2. Example XML scenario

IV. SYSTEM CONFIGURATION AND SCENARIO DESCRIPTION

D-Cloud executes system tests according to a scenario written in XML. By describing the scenario, the tester simultaneously executes various system tests. Figure 2 shows a complete example of a scenario statement. The scenario statement consists of four parts defining the test as follows:

- Description for hardware environment
- Description for software environment
- Definition of faults for injection
- Procedure of the entire test

A. Configuration for the hardware environment

The definition of the hardware environment is given in the “machineDefinition” element. Table II lists the contents of the “machineDefinition” element. Next, the definition of the hardware environment is described in the following. The “machineDefinition” element can hold multiple “machine” elements. The “machine” element includes five elements (name, cpu, mem, nic, and id), and these elements are needed for each definition of a hardware environment. The “name” element

TABLE II
MachineDefinition ELEMENTS

element name	contents
machine	unit of definition of hardware environment
name	name of hardware environment
cpu	number of CPU
mem	size of memory
nic	number of NIC
id	ID of an OS image

TABLE III
SystemDefinition ELEMENT

element name	contents
system	unit of definition of software environment
name	name of sonfware environment
host	unit of definition of a testing host
hostname	name of each host
machinename	used machine element
config	used file of environment configuration

defines the name of a hardware environment. The "name" element is used in the "systemDefinition" element for the configuration of a hardware environment. The "cpu" and "nic" elements represent the number of CPUs and NICs, respectively, and the "memory" element defines the allocation size of the memory. The "id" element designates the selection ID of an OS image to be booted. Eucalyptus provides each OS image with a unique ID, so that the tester specifies a provided ID. Since a virtual machine can be run with various numbers of CPUs, NICs, and memory sizes, the tester can boot OS images with different parameters. In the example scenario shown in Figure 2, "nodeA" is defined as "machineDefinition". "NodeA" has one CPU core, 512 MB of memory, and three NICs and uses the OS image designated id-1.

B. Configuration of the software environment

The definition of the software environment is given in the "systemDefinition" element. Table III lists the contents of the "systemDefinition" element. The "systemDefinition" element can have multiple "system" elements, each of which has a "name" element and multiple "host" elements. In addition, a "system" element is required for each defined software environment. The "name" element defines the name of a software environment and is used in the "testDescription" element for selection of target nodes for a system test (See Section IV-D for details). The "host" element contains three elements (hostname, machinename, and config), which are required for each definition of a tested host. The "hostname" element defines the name of a host, and the "machinename" element is selected from the "machineDefinition" element for the configuration of the hardware environment defined in the "machineDefinition" element. Using the name defined in the "machineDefinition" element, D-Cloud takes over the configuration of the corresponding hardware environment. The "config" element is used to select a file for a number of environmental parameters. The file defines installed applications and the network configuration in advance. The tester uploads the file to D-Cloud Frontend.

```
<system>
    <name>system-B</name>
    <host>
        <hostname>client</hostname>
        <machinename>clientset</machinename>
        <config>clientconf</config>
    </host>
    <host>
        <hostname>loadbalancer0</hostname>
        <machinename>balancerset</machinename>
        <config>balancerconf</config>
    </host>
    <host>
        <hostname>loadbalancer1</hostname>
        <machinename>balancerset</machinename>
        <config>balancerconf</config>
    </host>
    <host>
        <hostname>webserver0</hostname>
        <machinename>serverset</machinename>
        <config>serverconf</config>
    </host>
    <host>
        <hostname>webserver1</hostname>
        <machinename>serverset</machinename>
        <config>serverconf</config>
    </host>
</system>
```

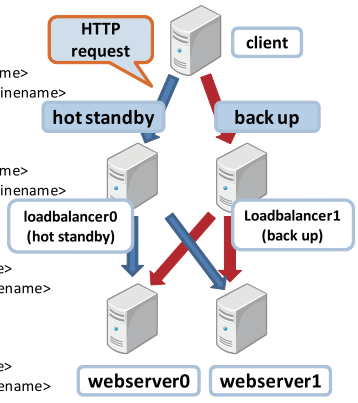

Fig. 3. Example scenario and configuration of the HA server

In the example XML scenario shown in Figure 2, the guest OS designated "node0" starts up using hardware environment "nodeA". After the guest OS has started up, a number of parameters on the guest OS are set according to the configuration file called "nodeconf", which is written in advance by the tester.

Figure 3 shows an example of a highly available server system (HA server system) as a distributed system. In the HA server system, HTTP requests from the client are normally forwarded to two back-end servers ("webserver0" and "webserver1") by "loadbalancer0" in a round-robin manner. In addition, if "loadbalancer0" is accidentally halted in an abnormal state, rather than "loadbalancer0", the reserved node "loadbalancer1" starts load balancing services. In this distributed system test, each configuration for all of the hosts is written. Guest OSes then start up according to the number of description of host elements. In the example shown in Figure 3, the "client" is booted as a client node, which issues HTTP requests. The "client" node is configured according to a hardware environmental element called "clientset" and a software configuration file called "clientconf". On the other hand, "loadbalancer0" and "loadbalancer1" start up as loadbalancers, which are configured according to a hardware environmental element called "balancerset" and a software configuration file called "balancerconf". Next, "webserver0" and "webserver1" start up as web servers and are also configured in the same manner as "client" and "loadbalancer[0-1]".

C. Configuration of fault injection

The definition of fault injection is given in the "injectionDefinition" element. Table IV lists the contents of the "injectionDefinition" element. The definition of fault injection is described in the following. The "injectionDefinition" element can have multiple "injection" elements, each of which has a "name" element and multiple "fault" elements. The "injection" elements are used to define the individual fault

injections. The “name” element defines the name of a fault injection and is used to specify the fault injection in the “testDescription” element. (See Section IV-D for details.) The fault element has four elements (“location”, “target”, “kind”, and “time”), which defines fault injections. The “location” element is used to select a target device, for example, a HDD, network or memory. The “target” element specifies a target device name, for example, “hda” or “hdb”, for HDDs, or “eth0” or “eth1” for NICs. The “kind” element indicates the selection of fault injection elements listed in Table I. The “time” element represents the duration time for fault injection. In the example shown in Figure 2, the fault injection is designated “injectionA”, and the fault injection for packet loss is executed into network device “eth0” for 40 seconds.

D. Configuration of test execution

The execution of the test is described in the “testDescription” element. Table V lists the contents of the “testDefinition” element. The “testDefinition” element can have multiple elements, each of which has “run”, “systemname”, “halt”, and multiple “script” elements. The “run” element is used for the individual definitions used to perform the test. If a tester wants to execute two different system tests, then the tester describes the two “run” elements individually. The “name” element defines the name of the system test to be performed. An output data file, the filename of which is the content of the “name” element, is dumped. The “systemname”element is selected from the name element of the “systemDefinition” element. The “halt” element indicates the finish time of the system test. The tester describes the “when”, which indicates the time from the start of booting a guest OS. The “script” element has four elements (“on”, “putFile”, “exec”, and “inject”), which enlarge on a execution. If a tester describes two or more nodes in a “systemDefinition” element, then the tester describes script elements for each node.

The “on” element is selected from the “hostname” element in the “systemDefinition” element. The “putFile” element is used to specify an uploaded file, the input files of which are required for the program to be tested. The uploaded file is used in the system tests on each node. The “exec” element is used to specify a script which should be executed in the test. The script has been uploaded in advance with “putFile”. The “inject” element is selected from the “name” element of the “injectionDefinition” element. If a tester wants to execute fault injection, which is defined in the “injectionDefinition” element, then the tester describes the injection name in the “injectionDefinition” element.

In the example shown in Figure 2, the name of the test is “testA”, and the test output obtained by the tester is designated “testA”. In the test environment designated “systemA”, the guest OS named “node0” uses “files” as inputs for the system test. The system test is then executed according to contents of “script” element, which defines a series of test procedures. During the system test, the injection defined as “injectionA” is executed 200 seconds after booting “node0”, and “node0” is then halted 400 seconds after the boot.

TABLE IV
`InjectionDefinition` ELEMENT

element name	contents
injection	unit of definition of fault injection
name	injection name
fault	configuration of injection
location	appointing device type
target	appointing device
kind	injection type
time	time of injection

TABLE V
`TestDefinition` ELEMENT

element name	contents
run	unit of definition of execution of test
name	name of test
systemname	appointing test environment
halt	finish timing of execution
script	unit of definition of execution of host
on	execution host
putFile	file transferred to guest OS
exec	execution of script written by tester
inject	execution of fault injection

V. D-Cloud operation

D-Cloud configures four components as four services.

- Portal service: D-Cloud provides a web portal to the tester. This web portal displays available OS images and provides the environment for data transfer.
- Configure test environment service: D-Cloud configures the test environment using the “machineDefinition” element and the “systemDefinition” element.
- Job execution service: D-Cloud executes a program test according to a given scenario using the “testDescription” element and the “injectionDefinition” element.
- Data management service: D-Cloud transfers test data given by the tester to appropriate nodes. Moreover, D-Cloud transfers logs and snapshots such as memory snapshot from test nodes to the tester after testing.

Figure 4 shows the D-Cloud procedure.

1) A tester describes a test scenario, configuration files, and scripts.
2) The tester uploads the test scenario, the configuration files, the script, and the files to D-Cloud frontend. If the tester wants to use OS images customized by the testers, the tester uploads these images to D-Cloud frontend.
3) D-Cloud frontend issues instructions for setting up the guest OSes for test to the Eucalyptus controller. If a tester has their own guest OS images, D-Cloud frontend transfers the OS images to the Eucalyptus controller. Otherwise, the tester must select a preconfigured OS image provided by Eucalyptus as a guest OS image to be transfered.
4) The Eucalyptus controller selects an available QEMU node to boot the guest OSes and to transfer the OS

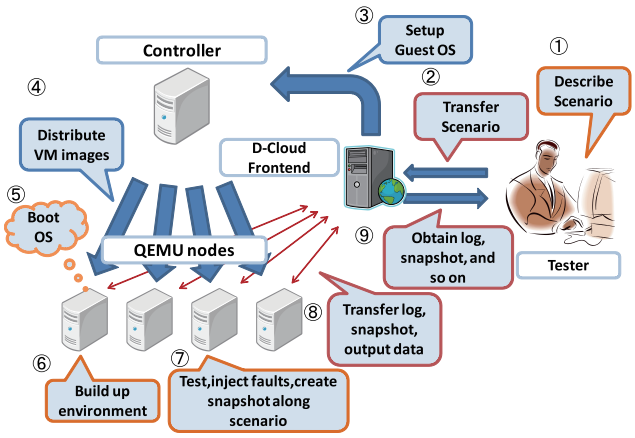

Fig. 4. Flow of D-Cloud

images to the selected QEMU nodes.

5) OS images are booted on the selected QEMU nodes.
6) D-Cloud frontend transfers the files used in the tests to be executed on each guest OS. Each guest OS then configures a test environment.
7) According to the test scenario, system tests and fault injection are executed and snapshots are taken.
8) After the system tests, each guest OS transfers the output data, logs, and snapshots to D-Cloud frontend.
9) The tester then downloads the output data, snapshots, and logs obtained through system testing.

VI. Related Research

A number of studies have carried out tests on multiple nodes in order to investigate dependability. GridUnit[4] executes a software test on multiple nodes using OurGrid[5] for distributed program tests. GridUnit uses JUnit and runs execution units defined by JUnit on multiple nodes. After test nodes are crashed and stopped, the test nodes cannot execute remaining program tests on the kernel layer. ETICS[6] provides automated test environments on a grid computing platform using Condor[7]. Although ETICS looks like the D-Cloud concept, ETICS does not provide the virtual machine environment unlike D-Cloud, ETICS cannot perform tests on the kernel layer, and restart using the snapshot image. Open Solaris Test Farm[8] uses a cloud computing environment. Open Solaris Test Farm enables user to create VM instances for program tests though a web portal and enables tests to be performed on these instances. However, since the functions that a tester can use on Open Solaris Test Farm are limited, the tester cannot configure test environments or inject faults.

A number of studies have injected faults in program tests. Although software fault injection by modifying the source codes to be tested has been proposed[9], the goal of the present study is to not modify the source codes for fault injection. In addition, a number of studies have considered fault injection using virtual machines, such as FAUmachine[10]. However, since these methods does not provide an automated test environment, the tester must configure the test environment manually.

VII. Conclusion and Future Research

In the present paper, we proposed the D-Cloud software test environment system, which can automatically configure test environments, execute tests, and automatically inject faults into hardware devices in a virtual machine. We have implemented a prototype D-cloud using Eucalyptus and QEMU. Using D-Cloud, a tester can execute a program test for a distributed system in appropriate environments according to a scenario written in XML. In addition, the tester can perform several tests by injecting faults into hardware devices in a virtual machine. The prototype D-Cloud can inject a number of faults into a HDD, NIC, or memory on a VM and can automatically execute a system test using a scenario file.

In the future, we intend to consider the reproducibility of the system test and to design a user interface for more easily writing scenario files using a web portal.

Acknowledgment

The present study was supported in part by the JST/CREST program entitled “Computation Platform for Power-aware and Reliable Embedded Parallel Processing System” in the research area of “Dependable Embedded Operating System for Practical Use”.

References

[1] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, “The eucalyptus open-source cloud-computing system,” in *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

[2] QEMU, open source processor emulator. [Online]. Available: http://www.qemu.org/

[3] Amazon elastic compute cloud (Amazon EC2). [Online]. Available: http://aws.amazon.com/ec2/

[4] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado, “Gridunit: software testing on the grid,” in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 779–782.

[5] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg, “OurGrid an approach to easily assemble grids with equitable resource sharing,” in *9th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 2862, June 2003, pp. 61–86.

[6] M.-E. Begin, G. D.-A. Sancho, A. D. Meglio, E. Ferro, E. Ronchieri, M. Selmi, and M. urek, “Build, configuration, integration and testing tools for large software projects: ETICS,” in *Rapid Integration of Software Engineering Techniques*, ser. Lecture Notes in Computer Science, vol. 4401, September 2007, pp. 81–97.

[7] F. J. González-Casta no, J. Vales-Alonso, M. Livny, E. Costa-Montenegro, and L. Anido-Rifón, “Condor grid computing from mobile handheld devices,” *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 2, pp. 18–27, 2002.

[8] Open solaris test farm. [Online]. Available: http://opensolaris.org/os/community/testing/testfarm

[9] S. Han, K. Shin, and H. Rosenberg, “Doctor: an integrated software fault injection environment for distributed real-time systems,” *Computer Performance and Dependability Symposium, International*, p. 0204, 1995.

[10] S. Potyra, V. Sieh, and M. D. Cin, “Evaluating fault-tolerant system designs using faumachine,” in *EFTS '07: Proceedings of the 2007 workshop on Engineering fault tolerant systems*. New York, NY, USA: ACM, 2007, p. 9.