Projet de Fin d'Études
2014 – 2015

[R|E]

D1.1 – Description of Work (DoW)

Fuzzing a VM scheduler

Participant:

- Alexandros TSANTILAS

Supervisors

- Fabien HERMENIER
- Ludovic HENRIO

Abstract

Inside an IaaS cloud, the VM scheduler is responsible for deploying the virtual machines to appropriate physical servers according to the SLAs. It is of critical importance as it has to make assignment of the virtual machines according to its specification and based on SLAs made with the customers. As environmental conditions and the clients' expectations evolve, the VM scheduler has to reconfigure the deployment accordingly.

However, implementing a VM scheduler that is correct and behaves according to its documentation is difficult and requires understanding of the infrastructure management capabilities and the pre-conditions related to each reconfiguration action, as well as combinatorial problems such as assignment and task scheduling.

The difficulties inherent in the implementation of a VM scheduler have led to defective implementations with severe consequences for both clients and providers. Therefore extensive testing is required so as to reveal as many bugs as possible. A more and more widespread testing technique to check complex software is fuzzing, which is based on generating random input data for a component to usually detect crashing situations or wrong results. However, the effectiveness of each fuzzer varies and depends mainly on the volume of code coverage, on whether or not it supports test-case reduction and of course on the number of bugs it finally reveals.

BtrPlace is a research oriented VM scheduler, which still has open bugs concerning correctness issues. Currently, some effort is made to fuzz BtrPlace but in addition to crashes, it is detected inconsistent behaviour with regards to the formal specification of some components. In this PFE, the goal is to go beyond these simple fuzzing techniques and propose novel exploration techniques. Therefore, the objectives of this PFE are:

- to establish a bibliography around fuzzing techniques, from generic to domain specific approaches.
- to propose novel exploration techniques.
- to implement them inside the current fuzzer of BtrPlace.

Table of Contents

1.General Project Description

Framework/Context

An Infrastructure as a Service (IaaS) cloud computing is a model according to which the user has the ability to provision computing, storage, or networking resources, provided by an organization. The client, who typically pays on a per-use basis, is able to develop and execute whatever software he wants, either it is an operating system or an application. He doesn't control the infrastructure of the cloud, but he has total control of system, storage, computing and networking operations. He is able to define his memory, computing power, storage volume and operating system requirements.
The hardware resources are typically provided to the user as virtual machines. The provider is the only responsible for housing, running and maintaining the equipment. The main features of an IaaS cloud are the dynamic scaling, the utility computing service and billing model and the policy-based services.

The VM scheduler is one of the most important elements for the good functioning of an IaaS cloud. At first, the clients demand their requirements based on the provider offerings. Then, the scheduler is expected to take decisions that are aligned with its theoretical behaviour and corrective actions on the deployment on the event of failures, load spike, etc and evolution of the clients' expectation. Therefore, its goal is to adjust the infrastructure's resources it uses, so as to accommodate varied workloads and priorities, based on SLAs with the customers. The amount of resources allocated and consumed is reflected on the cost, at the same time that providers are subject to penalties when the SLAs are not met in practice.

BtrPlace is a virtual machine scheduler for hosting platforms, that can be safely specialized through independent constraints that are stated by the users, in order to support their expectations [1]. On changes of conditions, it computes a new reliable configuration, according to some plans to reach it. Its aim is a more flexible use of cluster resources and the relief of end-users from the burden of dealing with time estimates.

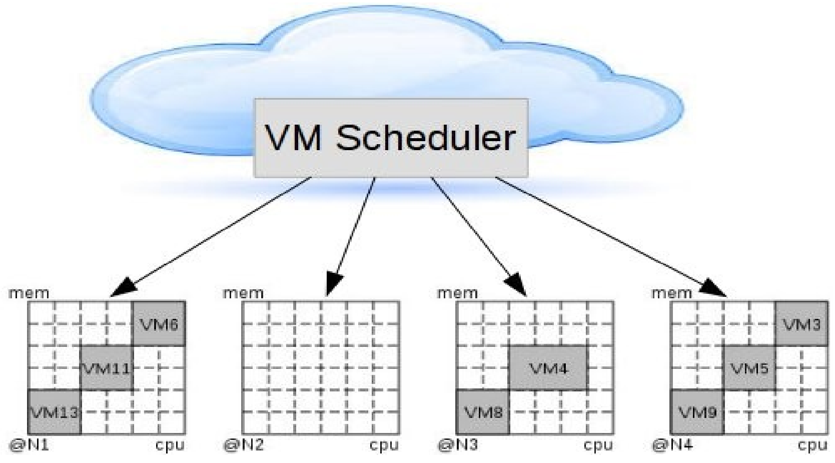

Figure 1: VM Scheduling example in BtrPlace. There are eight virtual machines assigned to four nodes.

However, it is not always easy to implement a VM scheduler that behaves according to its documentation. The developer should have a good understanding of the infrastructure management capabilities and the preconditions related to each reconfiguration action, in order to implement one, as well as master several families of combinatorial problems such as assignment and task scheduling . One example of preconditions is that no server can be turned off when VMs are running on it. In addition, to implement an SLA enforcement algorithm, the developer has to ensure that his code fits all the possible situations, by considering the implication of every possible VM state on its resource consumption.

Therefore, for the creation of a VM scheduler, extensive testing is required. For this reason, we use fuzz testing (known also as fuzzing), a more and more widely used technique for testing software systems. Fuzzing is an often automated or semi-automated technique, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. The most bugs in VM schedulers can be classified in three distinct categories, which are false-negative failures (invalid actions or VM scheduling), false-positive failures (a valid VM scheduling is prevented) and crashes.

Motivation

The difficulties that are applied in the implementation of a VM scheduler have led to the development of not correct schedulers and defective implementations. As a result, the SLAs are not always satisfied, with severe consequences for both clients and providers. For example, Nova is the component embedding the VM scheduler of the leading open source IaaS software stack OpenStack [2]. Despite a quality management system according to which the scheduler code is tested and the modifications are peer-reviewed before integration, 16 bugs are still currently open about correctness issues [3]. For instance, users reported that the VM scheduler computes the amount of consumed resources on servers incorrectly by taking crashed VMs into account. The same kind of bugs have been seen in the research oriented VM scheduler BtrPlace as well [4].

Even though more that eighty unit tests have been created, with a code coverage of 80% achieved and one thousand lines of code written for hand-written checkers, the BtrPlace VM scheduler's placement constraints are still bugged. This can result to a silent SLA violation, resource fragmentation, crashing reconfigurations or even runtime failures. A bug in a SLA enforcement algorithm tends to make clients of IaaS lose confidence in their providers. Likewise, a bug that exaggerates the amount of used resources reduces the gain for the provider. Delays in some major cloud computing infrastructures due to bad VM scheduling, are estimated to lead to millions of dollars loss, as it leads to less available resources and more energy consumption due to more occupied physical nodes.

The bugs that can occur in a VM scheduler like BtrPlace can be divided into the three main categories that are presented below, along with some representative examples of bugs that have been detected by the current fuzzer:

- **Runtime errors/Crashes:** Bugs of this kind result to a crash of the scheduler, which can be devastating as it is embedded in a larger system that stops working.
- **False-negative**: In these kind of bugs, the scheduler provides an invalid VM scheduling, that is not conforming to the constraints. Some examples of such bugs in the BtrPlace scheduler are the following:
  - The VMs are not allowed to have multiple states. If they do, then the states are necessarily in conflicts, which are not detected [5].
  - Shut-down a server hosting sleeping VMs should not be acceptable [6].

- Broken instances are considered to be consuming resources (Nova scheduler). These instances cannot be revived and should not be taken into account.
- **False-positive**: A valid scheduling is prevented or has problems, even though it is conforming well to the constraints. Such bugs are the following:
  - A virtual machine relocated with live migration is modelled using distinct time slices. Often, some VMs are counted twice with continuous capacity constraint [7].
  - When we request more resources that what our infrastructure (physical nodes) can provide, the problem sometimes fails when the constraint limiting the overbooking ratio is used with particular values [8]. For example, the constraint works fine with a ratio of 1.2 or 2, but not with a ratio of 1.4 or 1.5.

As the IaaS providers want to find as many bugs as possible to avoid the financial loss and the dissatisfaction of the customers, it is highly required to improve the bug detection techniques for their schedulers. The same applies for BtrPlace, that still has bugs remaining and for sure there are going to be more, unidentified by current fuzzer and unit testing.
The current fuzzer uses random test cases, that neither aim at a specific range in which they are probable to create more failures, nor use test-case reduction. In addition, no metrics such as a distance between bugs exist, in order to distinguish them. In fact, on reconfiguration the fuzzer produces a number between one and the number of physical nodes and gets the corresponding node. The constraints are also produced in the same random manner.

The virtual machines in BtrPlace have a defined lifecycle, as shown in the figure below. As we can see, the possible states are changed using an action on the virtual machine. The physical nodes can have only two states, online and offline.
When BtrPlace solves a problem, it considers that the state of the virtual machines stays unchanged expect if constraints force a change. BtrPlace only allows one state transition per VM. Then, it is not possible to have a VM in the Running state and relocate it twice, or put a ready VM in sleeping mode.

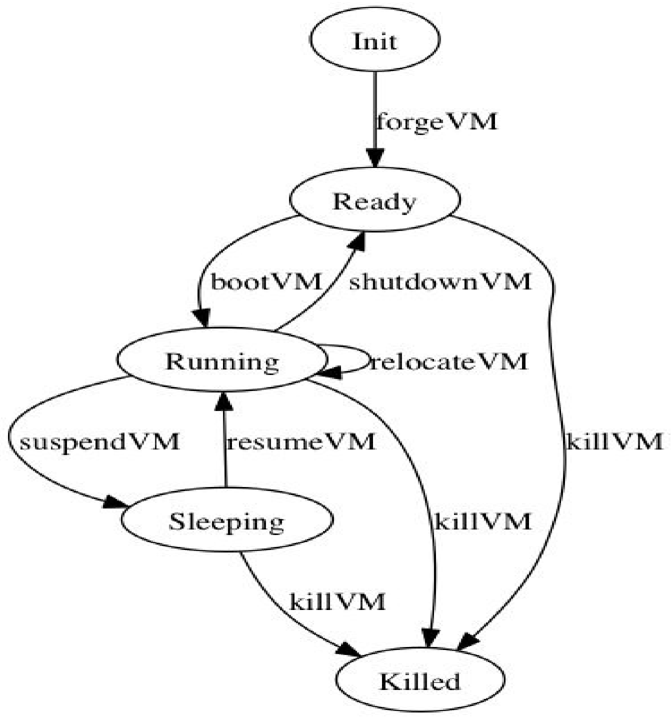

Figure 2: VM lifecycle in BtrPlace. There are five possible states that are changed on actions on VMs.

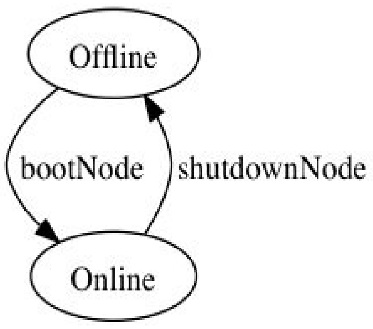

Figure 3: Lifecycle of a physical node

Concerning the test-case reduction, as the current fuzzer's algorithm is totally random and naïve, the same bug can come from just a small difference in the schedule of a test-case.
For example, let's assume that we have the following test-case, where Node 3 is banned, Node 2 hosts a sleeping VM3 and the input scenario asks Node 2 to shut-down. If Node 2 is allowed finally to shut-down, then there is a bug occurrence, as it is not allowed [6]. It is a false-negative bug. Let's also assume the same scenario with the sleeping VM3 on Node 3.

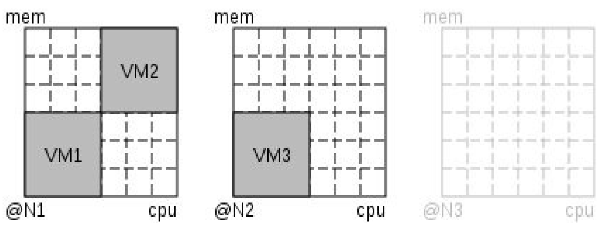

Figure 4: In this test-case 1, if N2 with sleeping VM3 is allowed to shut-down, it is a bug.

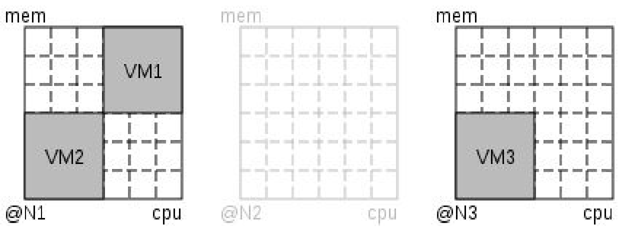

Figure 5: In this test-case 1, if N3 with sleeping VM3 is allowed to shut-down, there is bug. The two cases are similar.

Our fuzzer can produce both test-cases that lead to the same bug. As a result, it can produce more than one test-case that leads to the same bug, which is not desired. In fact, an infinite amount of different instantiations of the same bug can be generated. Whenever action sleeping starts before the termination of action shut-down, then the bug occurs. It can be avoided if we produce just one test-case for this scenario.

For this reason, we want to improve the current fuzzer's algorithm, so as to achieve a better code coverage, with more focus on test-case regions that can produce more bugs. We also want to reduce the test-cases, in a way that they trigger as many failures as possible.

Challenges

During the project, we have to overcome quite a few challenges, such as the following:

- Maximize the code coverage of the fuzzer, making use of more efficient and more intelligent bug exploration techniques.
- Create few test-case scenarios that can identify the maximum number of bugs, instead of reporting numerous failure scenarios that hide the root causes. In this way, the bug-fixing procedure is facilitated.
- Identify distinct bugs, by using similarity or distance metrics for the bugs.
- Fault reports must be expressed in a way that assists the developer in fixing the problem and direct him to the faulty elements.

Goals

The main goal of this project to ease the debugging and implementation of correct VM schedulers regarding their expected behaviour, by providing a suitable and clever fuzzing technique that can discover as many bugs as possible.
In order to achieve the above, the project objective is:

- to establish a bibliography around fuzzing techniques, from generic to domain specific approaches and conclude which can be used for the fuzzing of the BtrPlace.
- to propose novel exploration techniques for a better discovery of bugs in the BtrPlace.
- to implement them inside the current fuzzer of BtrPlace.

2. State of the Art

Recently, there are a lot of solutions that have been proposed for more efficient and useful fuzzers. Some state-of-the-art approaches are the following:

Test-Case Reduction

Before a bug can be reported, the circumstances leading to it must be narrowed down. The most important part of this process is test-case reduction: the construction of a small input (minimal test case) that triggers the compiler bug. In fact, this technique seeks to find the difference between two bugs and decide if they are the same or distinct ones. This may be done manually, or using software tools, where parts of the test are removed one by one until only the essential core of the test case remains.

The existing approach to automated test-case reduction is the Delta Debugging (dd) algorithm [9]. Its objective is to minimize the difference between a failure-inducing test case and a given template. The ddmin algorithm is a special case of dd where the template is empty and therefore its goal is to minimize the size of a failure-inducing test case.
Ddmin heuristically removes contiguous regions (called as chunks) of the test in order to generate a series of variants. Those that do not trigger the desired behaviour are called unsuccessful variants and are discarded, contrary to successful variants that are used as the new basis for producing other variants. If there can't be generated any successful variants from the current basis, the chunk size is decreased. The algorithm terminates when the chunk size cannot be further decreased and so no more successful variants can be produced and the last successful variant that was produced is the result. The failure inducing inputs are isolated automatically by the dd algorithm, by systematically narrowing down failure-inducing circumstances until a minimal set remains.

Another similar approach is the HDD (hierarchical delta debugging algorithm), at which the original dd algorithm is applied to each level of a program's input [10]. It exploits input structure to minimize failure-inducing inputs and manages to speed up the dd algorithm.

Backward dynamic slicing has also been proposed to guide programmers in the process of debugging by focusing the attention of the user on a subset of program statements which are expected to contain the faulty code [11]. The backward dynamic slice of a variable at a point in the execution trace includes all those executed statements which affect the value of the variable at that point. Therefore, they are able to contain the faulty statement in most of the cases and are quite small compared to the number of executed statements.

Taming fuzzers

Fuzzers can be frustrating to use, as they indiscriminately and repeatedly find bugs that may not be severe enough to fix right away. Therefore, an obvious drawback of large random test cases is that they contain much content that is probably unrelated to the bug and it are difficult to debug.

It is proposed to order test cases that trigger failures such that diverse, interesting test cases that trigger distinct bugs are highly ranked and are presented early in a list [12]. This can be achieved if we tame a fuzzer by adding a tool to the back end of the random-testing

workflow and using techniques from machine learning to rank the test cases. A fuzzer tamer can estimate which test cases are related by a common fault by making an assumption: the more “similar” two test cases, or two executions of the compiler on those test cases, the more likely they are to stem from the same fault. A distance function maps any pair of test cases to a real number that serves as a measure of similarity.
If we first define a distance function between test cases that appropriately captures their static and dynamic characteristics and then sort the list of test cases in furthest point first (FPF) order, then the resulting list will constitute a usefully approximate solution to the fuzzer taming problem. We can lower the rank of test cases corresponding to bugs that are known to be uninteresting.
Information retrieval tasks can often benefit from normalization, which serves to decrease the importance of terms that occur very commonly, and hence convey little information. Before computing distances over feature vectors, we normalized the value of each vector element using tf-idf.

Directed Automated Random Testing

On the one hand unit testing is very hard and expensive to perform properly, even though it can check all corner cases and provide 100% code coverage. Yet, it is also well-known that random testing usually provides low code coverage and is not checking the corner cases where bugs that are causing reliability issues are typically hidden.

For this reason, a new tool for automatic software testing named DART is proposed [13]. It combines the three following main techniques:
- automated extraction of the interface of a program with its external environment using static source-code parsing
- automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in
- dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct the execution along alternative program paths.

DART is able to dynamically gather knowledge about the execution of the program. Starting with a random input, a DART-instrumented program calculates during an input vector for the next execution, during each one. This vector contains values that are the solution of constraints gathered from statements in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path, thus performing a directed search. The goal is to explore all paths in the execution tree.

Feedback-oriented random test generation

This technique improves random test generation incorporating feedback obtained from executing test inputs as they are created [14]. It builds inputs incrementally by randomly selecting a method call to apply and finding arguments from previously-constructed inputs.
The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique then outputs a test suite consisting of unit tests for the classes under test. From them, passing tests can be used to ensure that code contracts are preserved across program changes and failing tests point to potential errors that should be corrected. While it retains the scalability and implementation simplicity of random testing, it also avoids the generation of redundant and meaningless inputs, and is therefore competitive with systematic techniques.

A method sequence, or simply sequence, is a sequence of method calls. It builds sequences incrementally, starting from an empty set of sequences. As soon as a sequence is built, it is executed to ensure that it creates non-redundant and legal objects, as specified by filters and contracts.

Grammar-based whitebox fuzzing

The current effectiveness of whitebox fuzzing [15] is limited when testing applications with highly-structured inputs [16]. The goal is to enhance whitebox fuzzing of complex structured-input applications with a grammar-based specification of their valid inputs. Based on experiments, it is proven that grammar-based whitebox fuzzing generates higher-quality tests that examine more code in the deeper, harder-to-test layers of the application.

Swarm testing

Swarm testing is a novel way to improve the diversity of test cases generated during random testing, that contributes a lot to an improved coverage and fault detection [17]. In swarm testing, instead of including all features in every test case, a large "swarm" of randomly generated configurations, is used. Each of them omits some features like API calls or input features, with configurations receiving equal resources.
This technique has several important advantages. First of all it is low cost and secondly it reduces the amount of human effort that must be devoted to tuning the random tester. Based on experience, existing random test case generators already support or can be easily adapted to support feature omission.
Initially, when features appear together only infrequently over a test configuration C_i this may lower the probability of finding the "right" test for a particular bug, but does not preclude it. Second, since other features will almost certainly be omitted from the few C_i that do contain the right combination, the features may interact more than in C_D, thus increasing the likelihood of finding the bug.

Differential testing

In a recent work [19], a randomized test-case generator using differential testing was also proposed for finding bugs in C compilers. Differential testing consists an additional mature random testing method for large software systems [18]. It complements regression testing based on commercial test suites, comparing them with locally developed tests, during product development and deployment. Given two or more available systems to the tester, they are presented with an exhaustive series of mechanically generated test cases and if the results differ, one of them is a candidate for a bug-exposing test.

Bottom line

Despite the fact that all of these techniques seem quite effective and interesting, not all of them can be implemented in the BtrFuzzer. From a first point of view, the best candidate solutions would be a dd algorithm for test-case reduction, a ranked list of bugs for the discovery of distinct ones and swarm testing for better code coverage. A sort of directed automated random fuzzing technique could also be applied, provided that we can find a region where we can deduct that a lot of failure inducing test-cases are contained.

3. Workplan, Tasks and Milestones

Workplan and tasks

The basic workplan for the realization of the project is shown in the figure below. In fact, it consists of three main phases, composed from smaller tasks, followed by the creation of a deliverable document.

	Task name	17/11-28/11	01/12/2014 – 09/01/2015	12/01/2015 – 25/02/2015
Definition phase	Get to grips with the topic Study of related work	17/11-28/11		
Analysis phase	Understand BtrPlace & understand existing fuzzer		01/12-10/12	
	State-of-the-art proposals		11/12-19/12	
	Propose first improvement		5/1-9/1	
Implementation phase	Implementation			12/1-30/1
	Further improvement & final report preparation			16/2-24/2

In more detail, the individual tasks of each phase are organized as following. In our case, a full-time working week means 5 full-days of work and part-time working week is 3 half-days of work.

Problem Definition and Understanding phase (17/11 – 28/11)
For this phase one week in parallel with exams and one week of full-time are dedicated. The main objective at the end of this phase is the writing of the Description of Work deliverable. The subtasks are the following:

1) Getting to grips with the topic.
   This task basically contains the understanding of the framework and the context of the project, its main challenges and goals and the motivation for working on it.
2) Reading some related work for tackling relevant problems.
   The studying of relevant published work and papers to tackle similar problems is the essential part of this task.

Analysis phase (01/12 – 09/01)
For this phase one week of full-time work and three weeks of part-time work are dedicated. The main objective is the deeper understanding of the project and the tools and software used, the analysis of the problem and comparison with solutions proposed for relevant problems and the proposal of a feasible solution for this case. At the end of this phase a mid-term deliverable is going to be prepared with the proposal that is going to be implemented. The individual tasks are the following:

1) Understanding how the BtrPlace VM scheduler works – Installation & hands-on experience (full-time 1/12- 3/12).
   The basic goal is to understand how the scheduler reconfigures the deployment and assigns VMs to the physical nodes and the relevant mechanisms. Installation of the scheduler and some first tests using the existing fuzzer.
2) Understanding of how the existing fuzzer of the BtrPlace scheduler works (full-time 4/12-5/12 and part-time 8/12 – 10/12).
   It requires the comprehension of the current BtrPlace fuzzer algorithm, by running into its implementation code. Find some corner points of the scheduler that need more extensive testing and that could hide bugs.
3) Understanding of state-of-the art proposals and checking if and how they can work in the BtrPlace scheduler (part-time 11/12-19/12).
   Analysis of some proposals in recent papers and comparison between them. Check which of them can be feasible solutions for the BtrPlace fuzzer.
4) Propose a first improvement on the BtrPlace fuzzer, combining some state-of-the-art proposals and adapting them to our project (part-time 5/1 – 9/1).
   Based on a proposal for fuzzing or a combination of more, propose a feasible solution for our case.

Implementation phase (12/01 – 25/02)
As the appearance of obstacles and difficulties during this phase is highly possible, the time period required for the implementation task can be reconfigured and extended. At the end of this period there are scheduled exams, so the progress will slow down. No work is scheduled during the exam period (which is between 2/2 and 13/2). However, it is possible that this task will start earlier if the previous tasks are evolving well, or even in parallel. In case of difficulties, there will be an extra meeting arranged with the supervisors of the project. At the end, the final report will be delivered and a presentation will be given.
1) Implementation of the proposed solution – improvement of the existing fuzzer of the BtrPlace (part-time 12/1-30/1).
   Integration of the proposed solution in the fuzzer of the BtrPlace.
2) Discussion of possible further improvement. A more mature solution proposed in theory.
3) Writing of the final report – Results – Evaluation (full-time 16/2 – 24/2).
   Preparation of the final document that contains the results, an evaluation of them and topics for future improvement.

Deliverables

After the completion of each phase, a deliverable is sent to the supervisors and the reviewer of the project. The final report will be accompanied by a presentation on the work done. The exact dates, as estimated currently, are shown below:

Deliverables	Date
Description of Work	28/11/2014
Problem Analysis (Mid-term deliverable)	09/01/2015
Final Report	25/02/2015

4. Bibliography

[1] Hermenier, Fabien, Julia Lawall, and Gilles Muller. "Btrplace: A flexible consolidation manager for highly available applications." *IEEE Transactions on dependable and Secure Computing* (2013): 1.
[2] OpenStack Nova: http://nova.openstack.org/
[3] Nova open bugs: https://bugs.launchpad.net/nova/+bugs?field.tag=scheduler
[4] BtrPlace bugs: https://github.com/btrplace/scheduler/issues
[5] https://github.com/btrplace/scheduler/issues/43
[6] https://github.com/btrplace/scheduler/issues/25
[7] https://github.com/btrplace/scheduler/issues/12
[8] https://github.com/btrplace/scheduler/issues/18
[9] Zeller, Andreas, and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." *Software Engineering, IEEE Transactions on* 28.2 (2002): 183-200.
[10] Misherghi, Ghassan, and Zhendong Su. "HDD: hierarchical delta debugging." *Proceedings of the 28th international conference on Software engineering*. ACM, 2006.
[11] Agrawal, Hiralal, Richard A. DeMillo, and Eugene H. Spafford. "Debugging with dynamic slicing and backtracking." *Software: Practice and Experience* 23.6 (1993): 589-616.
[12] Chen, Yang, et al. "Taming compiler fuzzers." *ACM SIGPLAN Notices*. Vol. 48. No. 6. ACM, 2013.
[13] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.
[14} Pacheco, Carlos, et al. "Feedback-directed random test generation." *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007.
[15] Whitebox testing: http://en.wikipedia.org/wiki/White-box_testing
[16] Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin. "Grammar-based whitebox fuzzing." *ACM Sigplan Notices*. Vol. 43. No. 6. ACM, 2008.
[17] Groce, Alex, et al. "Swarm testing." *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.
[18] McKeeman, William M. "Differential testing for software." *Digital Technical Journal* 10.1 (1998): 100-107.
[19] Yang, Xuejun, et al. "Finding and understanding bugs in C compilers." *ACM SIGPLAN Notices*. Vol. 46. No. 6. ACM, 2011.