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 VMs to appropriate physical servers according to the SLAs. 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 a VM scheduler have led to defective implementations with severe consequences for both clients and providers. Fuzzing is a software testing technique to check complex software, that is based in generating random input data for a component to usually detect crashing situations or wrong results.

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 a 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 model is a resource provisioning model in which an organization outsources the equipment used to support operations, including storage, hardware, servers and networking components. The service provider owns the equipment and is responsible for housing, running and maintaining it. The client typically pays on a per-use basis. 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 the cornerstone of the good functioning of an IaaS cloud. The provider makes his offerings and the clients demand their requirements based on the features described in its documentation. The scheduler is then expected to take decisions that are aligned with its theoretical behaviour and reconfigure the deployment according to the evolution of environmental conditions (failures, load spike, etc) and the clients' expectation. Therefore, its goal is to adjust the infrastructure resources it uses, so as to accommodate varied workloads and priorities, based on SLAs with the customers. The cost reflects the amount of resources allocated and consumed, minored by the providers' penalties when the SLAs are not met in practice.

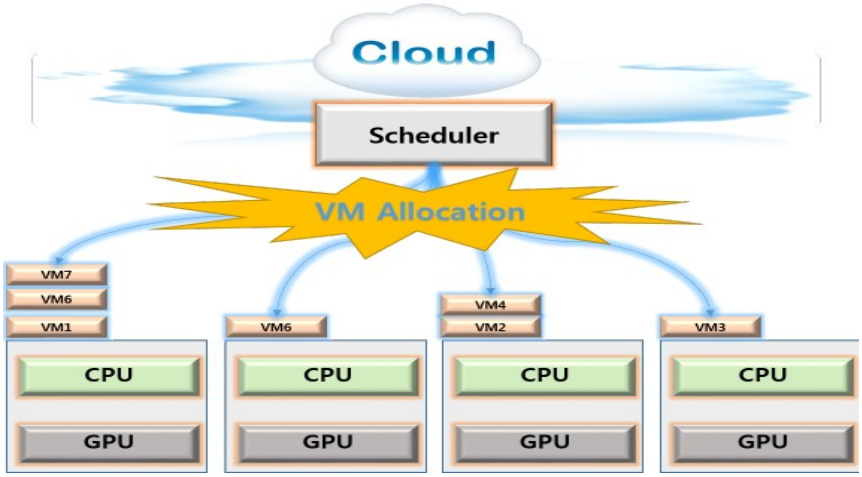

Figure: VM Scheduling example in a GPGPU Cloud

BtrPlace is a virtual machine scheduler for hosting platforms, that can be safely specialized through independent constraints to support the user's expectations [1]. It acts as an infinite control loop, which performs a globally optimized placement according to cluster resource usage and scheduler objectives. Relying on an encapsulation of jobs into VMs, it enables to implement finer scheduling policies through cluster-wide context switches: permutations between VMs present in the cluster. It results to a more flexible use of cluster resources and relieve end-users from the burden of dealing with time estimates.

Implementing a VM scheduler that is correct, i.e. that behaves according to its documentation, is however difficult. To implement a reconfiguration action, a developer must understand the infrastructure management capabilities and the pre-conditions related to each reconfiguration action. For example, one of the preconditions is that no server can be

turned off when VMs are running on it. To implement an SLA enforcement algorithm, the developer must also master several families of combinatorial problems such as assignment and task scheduling and ensure its code fits all the possible situations. For example, he must consider 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 inherent in the implementation of a VM scheduler have led to defective implementations 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. The scheduler code is tested and every modification must be peer-reviewed before integration. Despite this quality management system, users reported that the VM scheduler computes the amount of consumed resources on servers incorrectly by taking crashed VMs into account and that the algorithm that forces the collocation of VMs is broken when some of the VMs have been resized. Therefore, 8 bugs are still currently open about correctness issues and the same kind of bugs have been seen in the research oriented VM scheduler BtrPlace.

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.

The bugs that can occur in a VM scheduler like BtrPlace can be divided into 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**
- **False-negative**: In these kind of bugs, the scheduler provides an invalid VM scheduling, that is not conforming to the constraints.
  - Multiple future states for VMs, when states are necessarily in conflicts but not detected. Even worse, previous states are ignored.
  - Shutdown a server hosting sleeping VMs should not be acceptable.
  - Broken instances are considered to be consuming resources (Nova). 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.
  - Continuous counting of running VMs is wrong. This ignores the fact that a VM relocated with live-migration is modelled using slices that overlap on distinct nodes.
  - Problem gets solution with a ratio of 1, 1.2, 2 but fails with 1.5, 1.4.

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 remaning and for sure there are going to be more bugs, unidentified by current fuzzer and the unit tests. Therefore, 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 a few test-cases trigger as many failures as possible, so as to facilitate the bug-fixing procedure.

Challenges

During the project, we have to overcome quite a few challenges. Some of them are the following:

- The implementation must be consistent with its specification.
- The scheduler should always remain workable.
- The development lifecycle of the scheduler has to be retained.
- Fault reports must be expressed in a way that assists the developer in fixing the problem and direct him to the faulty elements.
- As far as the workplan is concerned, there should be an equivalence between research for fuzzing techniques and implementation of our solution.

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 (if not all). This can be achieved by maximizing the code coverage and by looking for bugs using specific input conditions that can discover the scheduler vulnerabilities. In addition, we aim to provide tools that exhibit a valuable feedback instead of reporting numerous and very specific failure scenarios that hide the root causes.

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 of them can be used for the fuzzing of the BtrPlace.
- to propose novel exploration techniques for a better discovery of bugs in the BtrPlace scheduler.
- 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. 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 text-case reduction is the Delta Debugging (dd) algorithm [2]. In fact, it seeks 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. Thus, ddmin's goal is to minimize the size of a failure-inducing test case.
Ddmin heuristically removes contiguous regions ("chunks") of the test in order to generate a series of variants. Unsuccessful variants are those that do not trigger the sought-after behaviour and are discarded. Successful variants, on the other hand, are those that trigger the desired behaviour and are used as the new basis for producing future variants. When no successful variants can be generated from the current basis, the chunk size is decreased. The algorithm terminates when the chunk size cannot be further decreased and no more successful variants can be produced. The final result is the last successful variant that was produced. The dd algorithm isolates failure causes automatically, by systematically narrowing down failure-inducing circumstances until a minimal set remains. Delta debugging has been applied to isolate failure-inducing program input.

Another similar approach is the HDD (hierarchical delta debugging algorithm), where the original dd algorithm to each level of a program's input, working from the coarsest to the finest levels [3].

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 [4]. 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 [5]. 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 causing reliability issues are typically hidden.

For this reason, a new tool named DART is proposed [6] for automatically testing software that combines three main techniques: (1) automated extraction of the interface of a program with its external environment using static source-code parsing; (2) automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in; and (3) dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths.
DART is able to dynamically gather knowledge about the execution of the program in what we call a directed search. Starting with a random input, a DART-instrumented program calculates during each execution an input vector for the next execution. This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path. The goal of DART is to explore all paths in the execution tree.

Feedback-oriented random test generation

This proposed technique improves random test generation incorporating feedback obtained from executing test inputs as they are created [7]. It builds inputs incrementally by randomly selecting a method call to apply and finding arguments from among 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 outputs a test suite consisting of unit tests for the classes under test. Passing tests can be used to ensure that code contracts are preserved across program changes; failing tests (that violate one or more contract) point to potential errors that should be corrected. It retains the benefits of random testing (scalability, simplicity of implementation), avoids random testing's pitfalls (generation of redundant or meaningless inputs), and is 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 is limited when testing applications with highly-structured inputs [8]. 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 exercise more code in the deeper, harder-to-test layers of the application under test.

Swarm testing

Swarm testing is a novel and inexpensive way to improve the diversity of test cases generated during random testing. Increased diversity leads to improved coverage and fault detection [9]. In swarm testing, the usual practice of potentially including all features in every test case is abandoned. Rather, a large "swarm" of randomly generated configurations, each of which omits some features, is used, with configurations receiving equal resources.
Swarm testing, in contrast, uses a diverse "swarm" of test configurations, each of which deliberately omits certain API calls or input features.
Swarm testing has several important advantages. First, it is low cost: in our experience, existing random test case generators already support or can be easily adapted to support feature omission. Second, swarm testing reduces the amount of human effort that must be devoted to tuning the random tester.
Initially, when features appear together only infrequently over 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

Regression testing is a type of software testing that seeks to uncover new software bugs, or regressions, after enhancements, patches or configuration changes, have been made to a system. The intent of regression testing is to ensure that changes such as those mentioned above have not introduced new faults.
Differential testing consists a form of random testing and is used as an additional mature testing method for large software systems [10]. It complements regression testing based on commercial test suites, comparing them with locally developed tests, during product development and deployment. Differential testing requires that two or more comparable systems be available to the tester, that are presented with an exhaustive series of mechanically generated test cases. If the results differ or one of the systems loops indefinitely or crashes, the tester has a candidate for a bug-exposing test.
In a recent work [11], a randomized test-case generator using differential testing was proposed for finding bugs in C compilers.

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 is consisted 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 midterm 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] Zeller, Andreas, and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." *Software Engineering, IEEE Transactions on* 28.2 (2002): 183-200.

[3] Misherghi, Ghassan, and Zhendong Su. "HDD: hierarchical delta debugging." *Proceedings of the 28th international conference on Software engineering*. ACM, 2006.

[4] Agrawal, Hiralal, Richard A. DeMillo, and Eugene H. Spafford. "Debugging with dynamic slicing and backtracking." *Software: Practice and Experience* 23.6 (1993): 589-616.

[5] Chen, Yang, et al. "Taming compiler fuzzers." *ACM SIGPLAN Notices*. Vol. 48. No. 6. ACM, 2013.

[6] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.

[7} Pacheco, Carlos, et al. "Feedback-directed random test generation." *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007.

[8] Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin. "Grammar-based whitebox fuzzing." *ACM Sigplan Notices*. Vol. 43. No. 6. ACM, 2008.

[9] Groce, Alex, et al. "Swarm testing." *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.

[10] McKeeman, William M. "Differential testing for software." *Digital Technical Journal* 10.1 (1998): 100-107.

[11] Yang, Xuejun, et al. "Finding and understanding bugs in C compilers." *ACM SIGPLAN Notices*. Vol. 46. No. 6. ACM, 2011.