Projet de Fin d'Études
2014 – 2015

[R|E]

D2
Final Report

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 this fact has 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 quite a few open bugs concerning correctness issues. The current fuzzer for discovering such bugs is not very efficient yet and new techniques should be applied for better code coverage and the creation of less, more effective test-cases that trigger distinct bugs.

For this reason, first of all in this document we present the currently open bugs in the BtrScheduler and divide them according to an appropriate classification. We then describe briefly how the current fuzzer works, underline its random behaviour and state the need for its improvement. Finally, we elaborate on our proposal, which includes a sort swarm testing exploiting the various configuration plans and a technique based on feedback from previous results and explain how they can be applied effectively to improve the current fuzzer's performance.

Table of Contents

1. Problem statement....4
Crashes....4
False-negative bugs....5
False-positive bugs....6

2. Current fuzzer's functionality....7
Random test-case generation....7
Probabilities for action transitions....8

3. Improved fuzzer....9
Proposed solutions....9
Diversity of configurations....9
Swarm testing....11

4.Bibliography....12

1. Problem statement

Context
schedulers
SLAs

Implementing a VM scheduler that is correct and behaves according to its documentation is usually difficult. It requires extensive 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. This can lead 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, so as to detect crashing situations or wrong results. However, the effectiveness of each fuzzer varies and depends on the volume of code coverage, the existence of test-case reduction and finally the number of bugs it reveals.

In the following subsections we examine the three main categories of bugs found in virtual machine schedulers. We elaborate on some representative bugs of the research-oriented BtrPlace scheduler [1][2], to which we will often refer in the future, as well as Nova that is the VM scheduler of the open source IaaS OpenStack [3][4]. What is really important to consider at this phase is the difficulty in observing and provoking each category of bugs.

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. In particular, in the BtrPlace scheduler we have observed the following bugs.

Bug #48 [5]:
This bug is in constraint “Spread” and ends up in an “Out of bounds exception” that terminates the execution of the scheduler. It can be reproduced in continuous time, during the scheduler's reconfiguration, when there are not only running VMs involved to this constraint. As we can see from the code triggering this bug, it adds to the VMs' array all the involved VMs, even though the array size is fixed by the number of the running VMs.

```
VM[] vms = new VM[running.size()];
int x = 0;
for (VM vm : cstr.getInvolvedVMs()) {
    vms[x++] = vm;
}
```

Figure 1: Code provoking a crash in BtrPlace

This kind of bugs is very easy to observe, as you can understand their existence because of the program's forceful termination.

The creation of such bugs is also relatively easy as there are already some ways to generate test-cases that cause the crash of a system. According to a recent research work [6], it is possible for a system to generate such test-cases on runtime using a combination of symbolic and regular program execution. In practice, this technique was applied to real code and created numerous corner test-cases, that produced errors ranging from simple memory overflows and infinite loops to complex issues in the interpretation of language standards. Furthermore, a classical crashing technique is to load the program with very large and possibly complex inputs.

False-negative bugs

In these kind of bugs, the scheduler provides an invalid VM scheduling, that is not conforming to the constraints. Therefore, such bugs provoke a violation of the SLAs between the provider and the customer. For example, we have observed the following behaviours:

Bug #43 [7]:
The VMs that have multiple states are definitely in conflicts that are not detected. In fact, in BtrPlace it is not allowed for a VM to be in multiple states. However, if we do set a virtual machine as both running and ready, we have a conflict which is not detected, as we can check from the following testing code:

```
Model mo = new DefaultModel();
Mapping map = mo.getMapping();
map.addOnlineNode(mo.newNode());
VM v = mo.newVM();
map.addReadyVM(v);
ChocoReconfigurationAlgorithm cra = new DefaultChocoReconfigurationAlgorithm();
Assert.assertNull(cra.solve(mo, Arrays.asList(new Ready(v), new Running(v))));
```

Figure 2: Code detecting the multiple state bug in BtrPlace

Bug #1012822 in the Nova scheduler [8]:
It is observed that broken instances are considered to be consuming resources, even though these instances cannot be revived and should not be taken into account. The Nova host manager simply adds all the resources for all the instances that are scheduled for that host, even though these instances are in error state. Therefore, instead of not existing at all, they consume resources. This can have a negative impact on the provider, that appears to have less resources than it actually has, which can possibly lead to a lack of hosting space for the customers that need to host their VMs and therefore provoke SLA violations.

Bug #1227925 in Nova [9]:
When an instance is terminated on a compute node, the resource tracker keeps the resources allocated for some time. Instead, it should remove the resources as soon as the instance is done being cleaned up. This can also lead to lack of hosting pace.

Bug #25 [10]:
In the BtrPlace we have also observed that it is possible to shut down a server that is hosting a sleeping virtual machine. This bug is in the constraint “no sleeping VMs on offline nodes”

and it occurs every time we try to shut down a server that hosts a sleeping virtual machine. Currently, it is only ensured that there will not be running VMs, although sleeping VMs should be considered when shutting down a physical node. This can be very negative for users that maintain a sleeping virtual machine, as their data can be lost if the physical node in which it is running is shut down. Therefore, it can provoke serious a SLA violation.
It is not difficult to realize this kind of bugs, which are observable after measuring and evaluating the result and comparing it to the expected one.
For instance, let's assume that we have an occurrence of bug #25. In this case, we observe the bug as soon as we realize that a host in which there was a sleeping VM is shut-down. The expected result would be for the host not to be shut-down but continue being turned-on.

But how easy is to create tests for this kind of bugs, before reporting an issue? Currently, using the existing random fuzzer for generating test-cases in the BtrPlace, it is not easy to detect them, as they are triggered randomly.

False-positive bugs

In these kind of bugs a valid scheduling is prevented or has problems, even though it is conforming well to the constraints. In this case, a test is marked as failed even in reality it should pass or if the functionality works properly. Similarly, automated testing can report an action that is provoking bug, even if this action is not possible at all. For example, the following bugs have been observed:

Bug #12 [11]:
In BtrPlace, some VMs are counted twice with the continuous capacity constraint. A virtual machine that is relocated with live migration is modelled using distinct time slices. Therefore, this bug, can occur on reconfiguration of the scheduler and migration of a virtual machine to another node. However, it can happen randomly, so it is actually difficult to reproduce. That can make the scheduler think that there are more resources allocated than in reality and therefore restrict the creation of new Vms.

Bug #18 [12]:
When we request more resources that what our infrastructure can provide, the problem sometimes fails when the constraint limiting the overbooking ratio is used with particular values. 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 we can check from the testing code shown below. This can prevent some valid configurations that the scheduler can have, even though they should be allowed. The occurrence of this bug is random and its cause is difficult to understand, as is detected with certain values only. This can lead to inability to host customer VMs and therefore provoke an SLA violation.

```
ChocoReconfigurationAlgorithm cra = new DefaultChocoReconfigurationAlgorithm();
cra.labelVariables(true);
cra.setVerbosity(1);
List<SatConstraint> cstrs = new ArrayList<>();
cstrs.add(new Online(map.getAllNodes()));
Overbook o = new Overbook(map.getAllNodes(), "foo", 1.5);
o.setContinuous(false);
cstrs.add(o);
cstrs.add(new Preserve(Collections.singleton(vm1), "foo", 5));
ReconfigurationPlan p = cra.solve(mo, cstrs);
Assert.assertNotNull(p);
```

Figure 3: Code detecting the overbooking ratio bug in BtrPlace

Bug #44 [13]:
We have observed that if a running virtual machine wants to migrate while other virtual machines want to boot, the "continuous among" constraint does not allow this action and therefore it appears to be restrictive. This bug is quite serious, as it prevents a valid migration of a virtual machine and it is difficult to reproduce, as it occurs randomly.

This kind of bugs is very difficult to observe, as the software developer should analyse them and deduct that the given configuration should be allowed by the program. The reproduction of such bugs is also quite difficult and often has to be based on very particular failure inducing test-cases, in regions that the programmer already knows that are likely to hide bugs. Therefore, they could be observed by intelligent fuzzers, that know where to search to this kind of bugs.

TODO: talk about different and similar bugs

2. Fuzz testing

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

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 [14]. 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 [15]. 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 [16] is limited when testing applications with highly-structured inputs [17]. 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 SWARM REF].
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 DIF REF], 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 DIF REF]. 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.

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 TAMING REF]. 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.

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.

TODO: How applied in schedulers in general. Why interesting?

3. Verifying BtrPlace

TODO: Verifying using fuzzing. Rearrange a bit.

Fuzz testing (known also as fuzzing) consists 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.

In order to find as many bugs as possible and solve the above mentioned problems in the BtrPlace, we have to implement an efficient fuzzer, that will be able to detect crashing situations or wrong results. However, before proposing our solution, it would be better to describe how the current fuzzer works, examine its weaknesses and explain why we need an improved one. Therefore, in the following subsections we prove the random test-case generation of the current fuzzer is very naïve and simplistic, that the transition probabilities are hard-coded and that it does not exploit the diversity of configurations for the scheduler.

TODO: catches also false-negative and false-positive

TODO: what parameters I ca change for testing

Totally random test-case generation

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 and reveal the test-cases that produce distinct bugs. For example, on migration of a virtual machine to a new physical node, the fuzzer produces a number between one and the total number of physical nodes and gets the result. The constraints are also produced in the same random manner.

In BtrPlace, the possible states are changed using an action on the corresponding element (virtual machine or physical node). When BtrPlace solves a problem, it considers that the

state of the element stays unchanged except if constraints force a change. BtrPlace only allows one state transition per VM.
The current fuzzer produces totally random test-cases in a way that:

- the scheduling of the events is random,
- the next state of a virtual machine or a physical node is basically random, according to their lifecycles shown below and some hard-coded probabilities,
- the action that is going to change the state of the element (that is forge, boot, shutdown, kill, relocate or suspend) happens also in the same random manner.

The issue described above yields to many test-cases for the same bug and probably prevents the creation of other test-cases that would lead to another distinct bug. Therefore, the code coverage of the current fuzzer is not satisfactory and should be improved.

The virtual machines and the physical nodes in the BtrPlace scheduler have a defined lifecycle, as shown in the figures below.

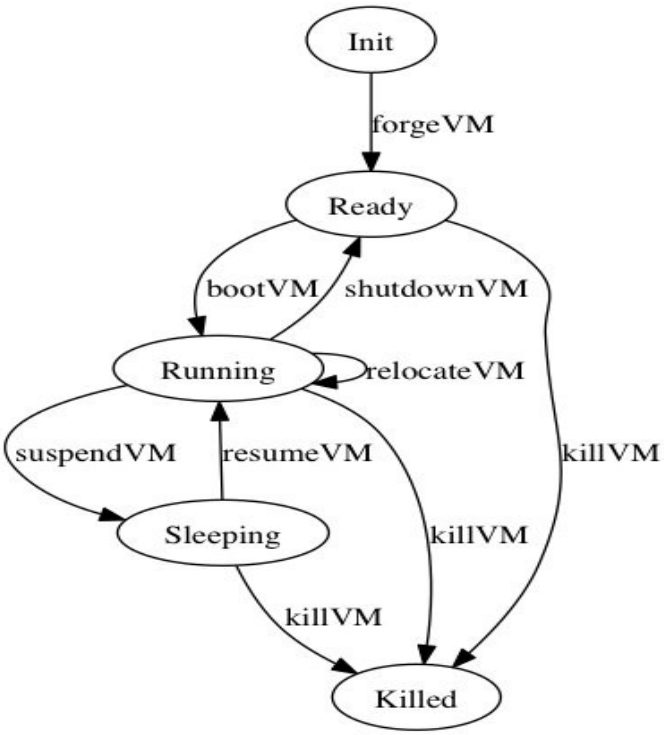

Figure 4: VM lifecycle in BtrPlace. There are five possible states that are changed on actions on VMs.

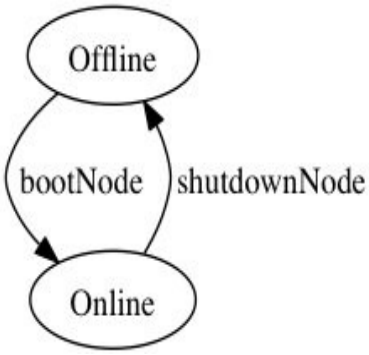

Figure 5: Lifecycle of a physical node

Static probabilities for action transitions

The current probabilities for the action transitions are hard-coded and very naïve. Below we can see these transitions for both the virtual machines and the physical nodes:

	initial	ready	running	sleeping	killed
ready	0.3	0.5	0.5	0	0
running	0.6	0.3	0.4	0.3	0
sleeping	0.1	0	0.2	0.8	0

Figure 6: VMs state transition probabilities

	initial	on	off
on	0.2	0.5	0.5
off	0.8	0.5	0.5

Figure 7: Physical nodes state transition probabilities

These probabilities are based on the experience of the BtrPlace software developer and his own judgement that the majority of bugs are triggered using these values. However, this calculation is too static and is not directed by the previous output of the fuzzer.

The main consequence of the static entries for the transition probabilities is that there is a limitation in detecting new bugs. The fuzzer is tending to produce similar test-cases that are detecting the same issues and are not able to understand that two bugs are the same.

TODO: Run tests using existing
extract tests and talk about similar bugs generated many times

4. Improved fuzzing for BtrPlace

It is very essential 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. Therefore, the main challenges we have to overcome are 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 and understand the similar ones.
- Fault reports must be expressed in a way that assists the developer in fixing the problem and direct him to the faulty elements.

The main aim of the proposed solutions is to confront the limitations imposed by the current random fuzzer and therefore detect more and different issues. Despite the fact that a lot of effective and interesting techniques have been proposed for similar projects, not all of them can be implemented in the BtrPlace fuzzer. In our case, the best techniques that can be used to improve our fuzzer are:

- a more directed fuzzer based on feedback from previous faulty configurations. The fuzzer should search for bugs in the regions where a lot of failure inducing test-cases

are contained by generating strongly different or similar text-cases and in this way exploit the diversity of configurations that we can observe in the BtrPlace scheduler. This can be achieved by a more intelligent way of calculating the transition probabilities.

- swarm testing in order to achieve better code coverage with a set of test-cases that trigger a lot of bugs by omitting some of the input features of the constraints.

Diversity of configurations

In order 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.
Regarding the diversity of the test-cases that the current fuzzer produces, it is not very satisfying. As the current fuzzer's algorithm is mostly random and naïve, the same bug can come from just a small difference in the schedule of a test-case. Instead, we should produce few test-cases for every buggy scenario and not more than one test-case that leads to the same bug.
There are two ways of producing different configurations. The first one is considering the different transitions of the VMs or the physical nodes and the second one the different possible time schedulings of two actions:

1. The possible transitions are:
   - VMs from running to ready, killed, sleeping or remain at the same state.
   - VMs from ready to running or remain at the same state.
   - VMs from sleeping to running or remain at the same state.
   - Physical nodes from offline to online.
   - Physical nodes from online to offline.
2. We can also change the relevant time schedule of two actions, checking what happens if they happen during the same time period or during strictly different time periods. If we have two transitions t_1 and t_2, we can have:
   - t_2 happening after t_1, for instance t_1 in [0:3] and t_2 in [2:5].
   - t_2 happening strictly after t_1, for instance t_1 in [0:3] and t_2 in [4:7].
   - t_2 happening before t_1, for instance t_2 in [0:3] and t_1 in [2:5].
   - t_2 happening strictly before t_1, for instance t_2 in [0:3] and t_1 in [4:7].

Now, let's examine these techniques with reference to the “No VMs on offline nodes” bug.

→ As observed, this bug is triggered when a physical node is going from online to offline and there is a virtual machine in sleeping mode. Therefore, in a “1 VM : 1 physical node” configuration that detects the bug, we have that:

- the initial state of the VM can be sleeping or running. Such probability is 2/4.
- the initial state of the physical node is online. Such probability is 1/2.
- the next state of the VM can be sleeping. Such probability is 1/4.
- the next state of the physical node is offline. Such probability is 1/2.

The total probability of detecting this bug with the current random fuzzer is therefore:
$\frac{2}{4}\cdot\frac{1}{2}\cdot\frac{1}{4}\cdot\frac{1}{2}=\frac{1}{32}$, so the possibility to find the bug is 1/32.

Indeed, we try to run first a test using the current fuzzer, only with the "no sleeping VMs on offline nodes" constraint. Indeed, our result is:

```
Bench.testNoVMsOnOfflineNodes: 100 test(s); 4 F/P; 0 F/N; 0 failure(s)
```

→ Therefore, we can see that it is difficult for a totally random scheduler to detect this bug, as this probability quite low. Instead, it would be much easier to detect it by a scheduler that focuses only on certain configurations. If we check for example exclusively on configurations in which the initial state of the physical nodes is online and the next is offline, then the probability of detecting the bug becomes:

$\frac{2}{4}\cdot\frac{1}{1}\cdot\frac{1}{4}\cdot\frac{1}{1}=\frac{1}{8}$, so the possibility to find the bug is 1/8.

Indeed, if we run a test on the same constraint, but allow the physical nodes go only from the online state to the offline, we have the following, much more impressive result:

```
Bench.testNoVMsOnOfflineNodes: 100 test(s); 35 F/P; 0 F/N; 0 failure(s)
```

→ The results would be even better if:

- we checked configurations in which the VMs are initially in running or sleeping state and then go to the sleeping state.
- given that s is the sleeping VM action and d is the ShutdownNode action, we tested the following schedules:
  - s happening strictly after d, for instance s in [0:3] and d in [2:5].
  - s happening after d, for instance d in [0:3] and s in [4:7].
  - s happening before d, for instance s in [0:3] and d in [2:5].

TODO: Mix 2 and 3 and say how techniques of section 2 can be applied to solve problems of section 3

Swarm testing in BtrPlace

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 [14]. 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. It is low cost and usually 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.

The objective of swarm testing is to examine the largest input range possible, something that can be achieved by two ways. The first one is by providing more general test-cases that omit some input features in order to test a more diverse set of inputs. The second one consists of creating test-cases for all the possible different regions, so that we can also check the system behaviour for a more diverse set of inputs.

Contrary to the second, the first one is more random and there is a possibility of providing low code coverage and not being able to check the corner cases of our scheduler configuration, where the bugs that are causing reliability issues are typically hidden. Therefore, in our case it is better to apply the second way, by examining all the different input regions and specifically these that hide the greater number of bugs.

Therefore, in our fuzzer we can generate test-cases taking into account:

- the different possible configurations in a way that we examine all the important transitions between the VM and physical node states and the different scheduling between the actions of two VMs or nodes.
- The probability transitions according to the calculation mentioned above.

5. 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] BtrPlace bugs: https://github.com/btrplace/scheduler/issues
[3] OpenStack Nova: http://nova.openstack.org/
[4] Nova open bugs: https://bugs.launchpad.net/nova/+bugs?field.tag=scheduler
[5] https://github.com/btrplace/scheduler/issues/48
[6] Cadar, Cristian, and Dawson Engler. "Execution generated test cases: How to make systems code crash itself." *Model Checking Software*. Springer Berlin Heidelberg, 2005. 2-23.
[7] https://github.com/btrplace/scheduler/issues/43
[8] https://bugs.launchpad.net/nova/+bug/1012822
[9] https://bugs.launchpad.net/nova/+bug/1227925
[10] https://github.com/btrplace/scheduler/issues/25
[11] https://github.com/btrplace/scheduler/issues/12
[12] https://github.com/btrplace/scheduler/issues/18
[13] https://github.com/btrplace/scheduler/issues/44
[14] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.
[15} Pacheco, Carlos, et al. "Feedback-directed random test generation." *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007.
[16] Whitebox testing: http://en.wikipedia.org/wiki/White-box_testing
[17] Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin. "Grammar-based whitebox fuzzing." *ACM Sigplan Notices*. Vol. 43. No. 6. ACM, 2008.

[18] Groce, Alex, et al. "Swarm testing." *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.
[19] McKeeman, William M. "Differential testing for software." *Digital Technical Journal* 10.1 (1998): 100-107.
[20] Yang, Xuejun, et al. "Finding and understanding bugs in C compilers." *ACM SIGPLAN Notices*. Vol. 46. No. 6. ACM, 2011.
[21] Chen, Yang, et al. "Taming compiler fuzzers." *ACM SIGPLAN Notices*. Vol. 48. No. 6. ACM, 2013.