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 open bugs concerning correctness issues. The current tool 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, in this document

- first of all, we describe the general context of cloud computing, virtual machine schedulers and service level agreements.
- secondly, we present the currently open bugs in the BtrPlace scheduler and divide them according to an appropriate classification.
- next, we describe how the current verification tool of the BtrPlace works, underline its random behaviour and state the need for its improvement.
- then, we describe fuzzing as a new effective approach of testing software and describe briefly the fuzzing techniques proposed recently and how they could be used to implement a fuzzer for a scheduler.
- finally, we elaborate on our proposal which includes improved techniques that exploit the various configuration plans effectively to discover more bugs. We also propose an algorithm that is supposed to improve a lot the current fuzzer's performance.

Table of Contents

1. Context & framework

Cloud Computing

According to the NIST definition of cloud computing [1], it consists a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. These resources can be either computing power, memory, networks, servers and storage, or applications and services that can be rapidly provisioned and released with minimal management effort or service provider interaction.

This cloud model is composed of five essential characteristics:

- on-demand self-service,
- broad network access,
- resource pooling, supporting multi-tenancy and providing location independence with dynamic assignment of different physical and virtual resources,
- rapid elasticity, due to the capability of scaling rapidly according to demand and
- measured service, with a usually pay-as-you-go or charge-per-use policy.

According to the type of service it provides, it can also be separated in three models:

- Software as a Service (SaaS), in which the end-user can use the provider's applications that are running on a cloud infrastructure. Such examples are GMail, Twitter and GitHub.
- Platform as a Service (PaaS), that consists programming languages, libraries, services and tools supported by the provider, so as to help the consumer deploy his own applications. Examples of this service type are Heroku, Google app engine, openshift and cloud foundry.
- Infrastructure as a Service (IaaS), that provides to the user the ability to provision computing resources, having control over the operating systems, storage and deployed applications (no control of the underlying cloud infrastructure though). Such examples are Amazon EC2 and Google compute engine.

In particular, an IaaS cloud computing is a model according to which the user can provision computing, storage, networking, or other 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. The client doesn't control the infrastructure of the cloud, but he has total control of system, storage, computing and networking operations. However, 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, while the user controls the resources provided, along with the deployed software. The main features of an IaaS cloud are the dynamic scaling, the utility computing service and billing model and the policy-based services.

Service Level Agreements

A service-level agreement (SLA) consists an agreement between the service user and a service provided. It defines the quality of service (QoS) provided, performance measurement, the responsibilities of the parties included, the customer duties, the pricing, warranties, termination and the penalties of the provider in case of violations. It is a part of a service contract where a service is formally defined, along with all the characteristics of the provided service. An SLA can depend from a lot of factors and is usually performance oriented. For this reason it has a technical definition in terms of performance indicators. Some examples are:

- Mean Time Between Failures (MTBF) = (Total up time)/(number of breakdowns);
- Mean Time To Repair (MTTR) = (Total down time)/(number of breakdowns);
- availability = MTBF/(MTTR + MTBF)

Usually customers require a good QoS and a particular guaranteed capacity (CPU, memory, bandwidth), latency and throughput. However, at the same time the provider opts to reach its personal objective, like reduced costs and reduced energy consumption.

Resource Management

Resource management in cloud computing refers to techniques for managing the cloud resources. It includes both allocating and releasing a resource when it is no more needed, preventing resource leaks and deals with resource distribution. The resource management in a cloud is achieved with the help of schedulers. The main goal of a scheduler is allocate the VMs in order to reach a certain awaited level of QoS required by the service users and is monitored continuously [2]. Obviously there is not a single or a perfect way to do this, but a series of different strategies with different final goals. The main concerns of a scheduler is to decide where to place the VMs and how many resources to allocate for them.

The schedulers are divided in static and dynamic ones. The schedulers that belong in the first category manage the scheduling of the VMs at the arrival time and therefore when a VM is allocated to a host, it can't be moved to another. On the contrary, a dynamic scheduler takes into account all the VMs (already placed and arriving ones) and reconfigures the scheduling, performing VM migrations. A VM migration is when a VM is moved from an initial host to a another one, along with its applications and data. Furthermore, a scheduler may allow overbooking or choose a conservative allocation. The former means that we can assign a VM to a host, even though there is not the memory or computing power asked for. This may result to performance losses with concurrent accesses to the VMs. The latter reserves for a VM exactly what it is asked for and doesn't allow overbooking.

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.

The BtrPlace VM scheduler

BtrPlace is a virtual machine scheduler for hosting platforms [3], that can be safely specialized through independent constraints that are stated by the users, in order to support their expectations. 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.

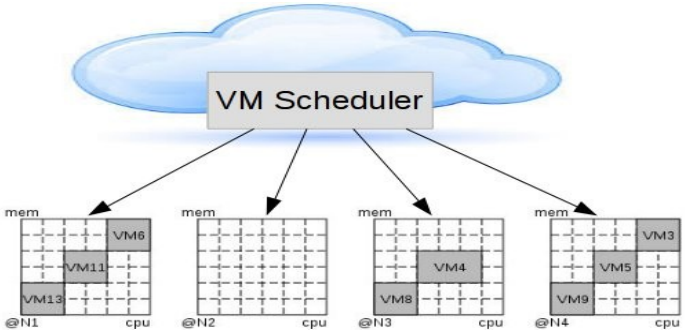

Figure 1: VM Scheduling example in BtrPlace.

For instance, let's consider the following reconfiguration example. Initially, we have sixteen VMs that are placed in the eight physical nodes, as shown in the following figure:

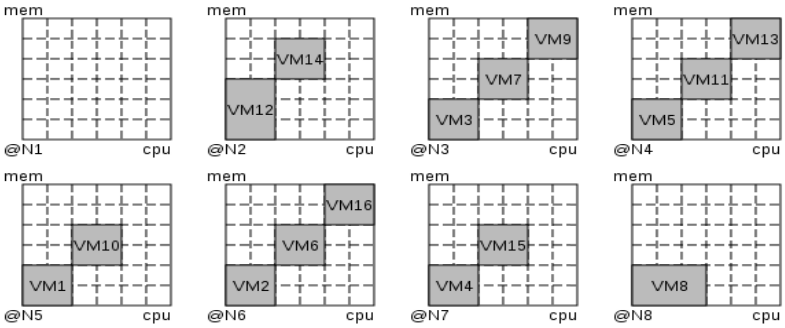

Figure 2: VMs initial configuration example.

Now, let's assume that we apply the following constraints. In fact, we want to separate VM12 and VM14 from running in the same node, ban VM5 from N4 and finally keep online at most one node among N1, N2 and N3.

```
spread({VM12, VM14})
ban(VM5, @N4)
maxOnline(@N[1..3], 1)
```

After the reconfiguration, the placement of the VMs on the physical nodes is as following:

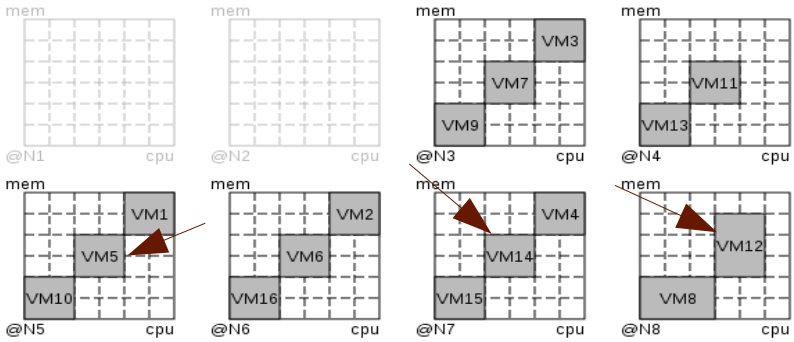

Figure 3: VMs final configuration after applying the constraints.

2. Motivation: Bugs in schedulers

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.

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 [4]. 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 [5]. 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 [6].

Two common techniques for bug finding and prevention are unit-testing and hand-written checkers. A checker is control code written directly inside the software component to check for the output correctness. However, 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. Bad VM scheduling in large cloud computing infrastructures can lead to delayed response times, less available resources and more energy consumption due to more occupied physical nodes.

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 BtrPlace scheduler, to which we will often refer in the future, as well as Nova scheduler. 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 in BtrPlace [7]:
In BtrPlace, the constraint “spread” means that we want to separate two or more VMs that reside at the same node. When setting the constraint “spread”, sometimes we end up in an “Out of bounds exception” that terminates the execution of the scheduler. This bug can be reproduced 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 4: 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 relatively easy as well, as there are already some ways to generate test-cases that cause the crash of a system.

According to a recent research work [8], it is possible for a system to generate such test-cases at 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-positive 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 in BtrPlace [9]:
A VM in BtrPlace can be in only one state, like running, sleeping, ready or killed. The VMs that have multiple states are definitely in conflicts. However, if we do set a virtual machine as both running and ready, we have a conflict which is not detected. This can be confirmed from the following test, where we try to solve the reconfiguration problem setting a new VM as both running and ready.

```
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 5: Code detecting the multiple state bug in BtrPlace

Bug #1012822 in the Nova scheduler [10]:
It is observed that corrupted, non-functional 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 a host. Therefore, instances that are in error state consume resources, instead of not existing at all. This has a negative impact on the provider, that appears to have less resources than it actually has. This can possibly lead to a lack of hosting space for the customers that need to host their VMs and a lack of revenue for the cloud provider.

Bug #25 in BtrPlace [11]:
BtrPlace allowed to shut down a server that is hosting a sleeping virtual machine, despite the fact that this action destroys the VM unexpectedly. 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 all its 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.

False-negative bugs

In these kind of bugs a scheduling that is viable in theory is disallowed by the scheduler. 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 #18 in BtrPlace [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. This can prevent some valid configurations that the scheduler can have, even though they should be allowed. As a result, the scheduler can't host some customer VMs and therefore an SLA violation is provoked. This bug occurs randomly and its cause is difficult to understand, as is detected with certain values only.

The bug can be reproduced from the testing code shown below, where the overbooking ratios tested for cpu are 1.5 and 5.

```
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(), "cpu", 1.5);
o.setContinuous(false);
cstrs.add(o);
cstrs.add(new Preserve(Collections.singleton(vm1), "cpu", 5));
ReconfigurationPlan p = cra.solve(mo, cstrs);
Assert.assertNotNull(p);
```

Figure 6: Code detecting the overbooking ratio bug in BtrPlace

Bug #12 in BtrPlace [13]:

Constraints in BtrPlace can provide either a discrete or a continuous restriction. In our case, a discrete restriction only focuses on the datacenter state by the end of the reconfiguration. On the other hand, a continuous restriction imposes limits even during the reconfiguration process.

In BtrPlace, with the continuous capacity constraint some VMs are counted twice, as a virtual machine that is relocated with live migration is modelled using distinct time slices. In the example shown below, VM1 is calculated twice until the end of the reconfiguration process.

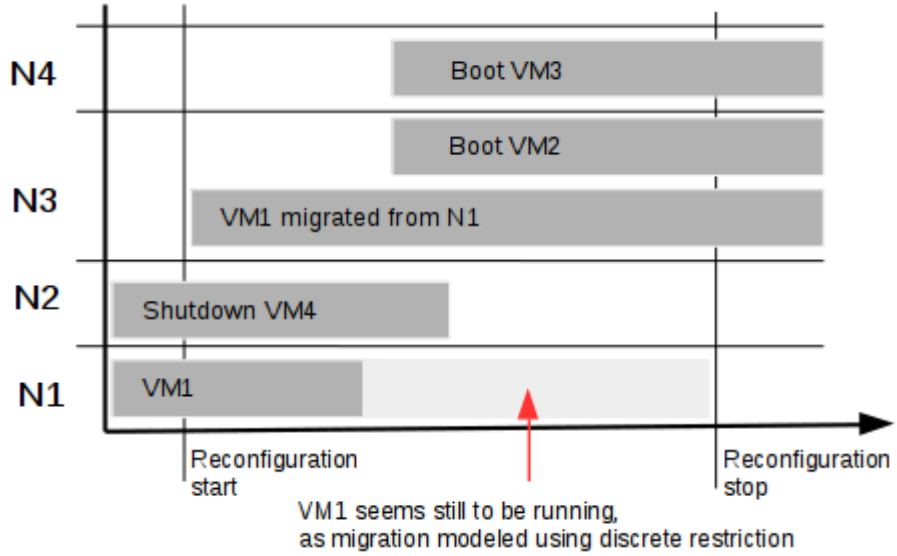

Figure 7: Demonstration of Bug #12.

This bug is observed sometimes during the reconfiguration of the scheduler and migration of a virtual machine from one host to another. Thus, the scheduler believes that there are more resources allocated than in reality. This bug is hard to reproduce as it requires a very specific setup in terms of VM management capabilities, type of constraint and type of restriction.

Bug #44 in BtrPlace [14]:

The constraint "among" forces a set of VMs to be hosted on single cluster of nodes among those that are available. This constraint is useful when the VMs are strongly communicating between each other. Indeed, the constraint will place the VMs on nodes that have a low network latency network between them. When the restriction is discrete, the constraint only ensures that the VMs are not spread over more than one cluster of nodes at the end of the reconfiguration process. When the restriction is continuous, if some VMs are already running on a group of nodes, it will not be possible to relocate the VMs to a new group of nodes.

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 over-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 in a very particular situation.

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 program input regions that the programmer already knows that are likely to hide bugs.

3. Problem: Limitations verifying BtrPlace using a fuzzer

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 the current automated bug detection tool and by unit testing.

The automated bug detection tool used currently for testing BtrPlace is using fuzz testing, an automated technique whose basic concept is generating random input data for testing a software component. The workflow of the current tool is the following:

- generate random input,
- test theoretical result using a model-checker,
- test implementation result,
- declare bug if the results mismatch,
- reduce the bugs as possible to ease understanding.

The current fuzzer has some useful features, like catching and distinguishing false-negative and false-positive bugs as well except for crashes. However, it has also important weaknesses, such as low code coverage and identification of the same bug multiple times. This means that it does not apply test-case reduction, a technique whose goal is to construct a minimal test case that triggers the bug.

For this reason, an improved fuzzer is needed. However, before proposing our solution, it would be useful to describe how the current fuzzer works. Thus, 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, that it does not exploit the diversity of configurations for the scheduler and that it does not distinguish different from similar bugs.

Totally random test-case generation

In the BtrPlace scheduler, the virtual machines and the physical nodes have a defined lifecycle, shown in the next page. 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.

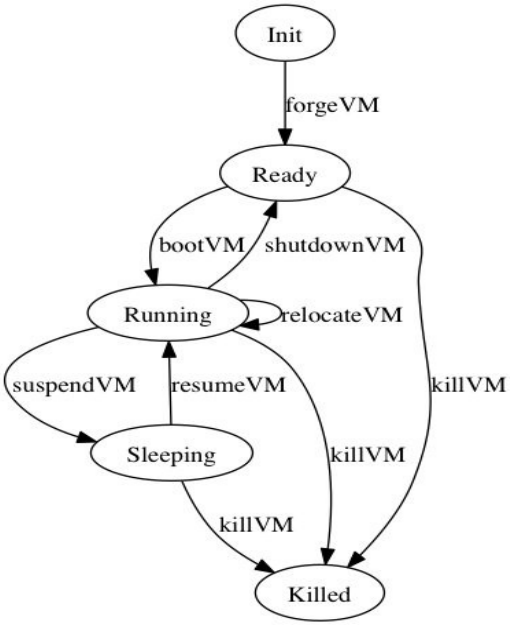

Figure 8: VM lifecycle in BtrPlace. There are five possible states that are changed on actions on VMs.

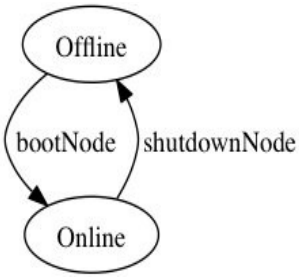

Figure 9: Lifecycle of a physical node

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.

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 10: VMs state transition probabilities

	initial	on	off
on	0.2	0.5	0.5
off	0.8	0.5	0.5

Figure 11: 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.

Similar bugs

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 [7]. It is a false-positive bug. Let's also assume the same scenario with the sleeping VM3 on Node 3.

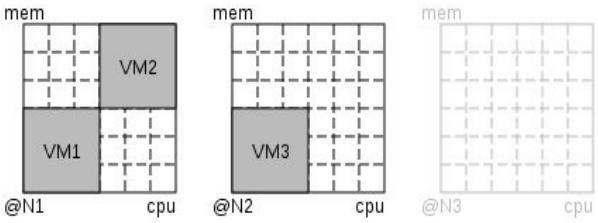

Figure 12: In this test-case if N2 with sleeping VM3 is allowed to shut-down, it is a bug.

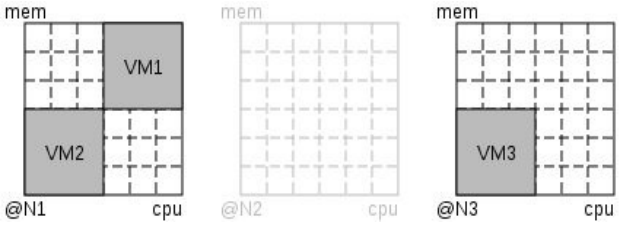

Figure 13: In this test-case, if N3 with sleeping VM3 is allowed to shut-down, there is bug. The two cases are similar.

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.

Now, let's run the fuzzer with the "noVMsOnOfflineNodes" constraint. We get the following final result:

```
Bench.testNoVMsOnOfflineNodes: 100 test(s); 5 F/P; 0 F/N; 0 failure(s) (6634ms)
```

As we can see, our fuzzer created 100 tests and discover 5 bugs. But if we examine two of the bugs that were detected, we can understand that these bugs are not all really different:

```
constraint: noVMsOnOfflineNodes
res: falsePositive
node#1: (vm#1)
node#0: -
READY vm#0
actions:
0:3 {action=shutdown(node=node#0)}
2:5 {action=shutdown(node=node#1)}
```

```
constraint: noVMsOnOfflineNodes
res: falsePositive
node#1: (vm#0)
node#0: -
READY vm#1
actions:
1:4 {action=shutdown(node=node#1)}
2:5 {action=shutdown(node=node#0)}
```

Figure 14: Bugs detected by our fuzzer for the "noVMsOnOfflineNodes" constraint
res = bug category
node#x: (vm) = VM-host matching (if VM in parenthesis, it is in sleeping mode)
actions = the actions operated during the reconfiguration

As we can see in the results, it is in fact the same bug. In the first case vm#1 on node#1 was in sleeping mode when node#1 was shutdown. In the second one it was exactly the same, except for the fact that it was vm#0 in sleeping mode in node#1.

4. State of the art: Fuzzing

Fuzzing techniques

In order to reveal as many bugs as possible, extensive testing is required. Unit testing is a very common technique to test software systems but it is very hard to perform properly, as we need to create lots of manually written tests to achieve a satisfactory code coverage. For this reason, fuzzing has become a more and more widespread testing technique to check complex software. It consists an often automated or semi-automated technique which is based on generating random input data for a component, so as to detect exceptions such as crashing situations, wrong results or potential memory leaks.
However, the effectiveness of each fuzzer varies and depends on:

- the volume of code coverage, which consists the percentage of code that is actually tested for bugs,
- the existence of test-case reduction, which provides:
  - the capability to distinguish bugs that are caused by the same or similar input,
  - bugs that are easy to understand,
- the number of total and distinguished bugs it reveals.

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. On the other hand, 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 tool for automatic software testing named DART has been proposed [15]. 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, and
- 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's goal is to dynamically gather knowledge about the execution of the program. Starting with a random input, a DART-instrumented program calculates 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.

This technique is very interesting in our study, as it can be effective if applied to construct a similar tool for fuzzing a VM scheduler. In fact, such a fuzzer could be designed to:

- get the set of constraints for the reconfigurations and the interface of the scheduler,
- create tests for all the constraints, covering as most of the possible inputs as possible,
- create more test-cases for regions that prove to hide more bugs, thus directing the test-case production in an automated way.

Feedback-directed random test generation

This technique improves random test generation incorporating feedback obtained from executing test inputs as they are created [16]. 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.

As with the previous technique, the fuzzer can be directed to produce more specific tests, that are likely to discover more bugs. In this case, the feedback-directed test-case generation could be used as following:

- create a few random test-cases for all the constraints,
- if there are bugs detected for some constraints, we can deduct that these constraints are buggy and it is useful to direct the fuzzer so as to generate more inputs and further examine these input regions.
- Generate more test-cases for the set of the buggy constraints.

Grammar-based whitebox fuzzing

The current effectiveness of whitebox fuzzing [17] is limited when testing applications with highly-structured inputs [18], as it rarely reaches parts of the application beyond the first processing stages, due to the enormous number of control paths in these early stages. 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. This is due to the fact that this algorithm creates fully-defined valid inputs, avoiding exploring the non-parsable inputs. By restricting the search space to valid inputs, grammar-based whitebox fuzzing can exercise deeper paths and focus the search on the harder-to-test, deeper processing stages.

Fuzzers for VM schedulers can exploit the main idea of this work and produce only valid test-cases. They can further restrict their number by creating test-cases based on the constraints set by the scheduler. Then, they will have the ability to focus the search on deeper stages and discover bugs that are deeper in the execution paths.

Swarm testing

Swarm testing is a technique that is used to improve the diversity of test-cases generated during random testing [19], contributing a lot to an improved code coverage and fault detection. 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.

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.
When implementing a fuzzer for a VM scheduler, 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.

Results optimization

Although fuzzers eliminate the need for creating lots of manually written tests for a sufficient code coverage, sometimes they can be frustrating to use. Very often, they repeatedly discover the same bug a lot of times, without realizing it. In addition, they can indiscriminately find bugs that may not be severe enough to fix right away. Therefore, a serious drawback of fuzzing and random test-case generation is that the results may include the same bug multiple times or contain much content that is probably unrelated to the bug, making it difficult to debug the software tested.

Taming fuzzers

To avoid this negative effect, it is proposed to order test cases that trigger failures in ranked list [22]. The list is ordered in a way that diverse, interesting test-cases that trigger distinct bugs are highly ranked and are presented early in it. This can be achieved if we tame a fuzzer by adding a tool to the backend 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.

This technique can be effectively used for VM schedulers as well. The algorithm can be based on the following statements:

- The bugs that are created by test-cases that are quite similar are highly possible to be triggered due to the same fault. Thus, these bugs should be lower in the ranked list.
- The bugs that are considered to be more important should be higher in the list. For instance, a good practice could be to rank higher crashes, as they force the termination of the scheduler. The next more important category should be false-positive bugs, that are allow a false reconfiguration and then false-negative that prevent a valid placement.

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 [23]. 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 [24]. 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 [25].

Although the above mentioned approaches are probably difficult to implement for VM schedulers, a sort of test-case reduction similar to the Delta Debugging approach could be used. In fact, the fuzzer should be able to distinguish the bugs that are duplicate and prevent them from being triggered more than once, by restricting the various test-cases created to have different enough inputs. In this context, a reducer could be created within a fuzzer, in order to distinguish test-cases with identical and similar inputs and remove them. As a result, in the end will remain only test-cases that trigger different bugs.

5. Solution: Improved fuzzer for BtrPlace

It is very essential to improve the current fuzzer's algorithm, so as to achieve a better code coverage, with higher 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 exploitation

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 satisfactory. 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 schedules 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.
In addition, given that s is the "sleeping VM action" and d is the "shutdown node" action, we would have better results if 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].

Identification of similar bugs

According to the inputs that are triggering two bugs, they can be denoted similar, different and strongly-different. Below, we elaborate on each of the three categories and illustrate this categorization by providing example inputs that have the following three dimensions:

$$\begin{Bmatrix} migration \\ VM\ transition \\ node\ transition \end{Bmatrix}$$

Similar bugs:

Bugs of this kind are triggered by very similar inputs. This means that such bugs are in fact a duplicates and should be reduced by a test-case reducer. For example, let's consider the following two inputs:

$$\begin{Bmatrix} no\ migration \\ running \rightarrow sleeping\ (0,3) \\ online \rightarrow offline\ (2,5) \end{Bmatrix} \quad \text{and} \quad \begin{Bmatrix} no\ migration \\ running \rightarrow sleeping\ (4,7) \\ online \rightarrow offline\ (5,8) \end{Bmatrix}$$

As we can see, the two inputs mentioned above are almost the same, as:

- the constraint is the same and requires that there are no VMs on offline nodes.
- the state of the nodes is changed from online to offline and the state of the VMs is changed from running to sleeping for both inputs.
- The state transition for the VMs is happening before the state transition of the nodes (not strictly before, as there is an overlapping time period).

These two inputs are going to trigger the same bug and more specifically it will be bug #25 of the BtrPlace scheduler (no sleeping VMs on offline nodes). Therefore, the fuzzer's reducer should understand the similarity of the bugs and therefore merge them into one.

Different bugs:

Such bugs are triggered by inputs that are different, even though one or more (but not all) inputs may be similar or the same. For example, let's have the following inputs:

$$\begin{Bmatrix} no\ migration \\ running \rightarrow sleeping\ (0,3) \\ online \rightarrow offline\ (2,5) \end{Bmatrix} \quad \text{and} \quad \begin{Bmatrix} no\ migration \\ sleeping \\ online \rightarrow offline\ (0,3) \end{Bmatrix}$$

These two inputs are going to trigger again the same bug, even though the scheduling is different and in the second input the state is remains the same (no transition from sleeping).
However, we can also have the following inputs, that are more different and produce different bugs:

$$\begin{Bmatrix} no\ migration \\ running \rightarrow sleeping\ (0,3) \\ online \rightarrow offline\ (2,5) \end{Bmatrix} \quad \text{and} \quad \begin{Bmatrix} migration \\ running \rightarrow sleeping\ (0,3) \\ online \end{Bmatrix}$$

Strongly-different bugs:

Bugs of this kind are triggered by entirely different inputs. For instance, let's have

$$\begin{Bmatrix} no\ migration \\ running \rightarrow sleeping\ (0,3) \\ online \rightarrow offline\ (2,5) \end{Bmatrix} \quad \text{and} \quad \begin{Bmatrix} migrate \\ running \\ online \end{Bmatrix}$$

As we can see, these inputs are entirely different and therefore produce different bugs. In fact, the second one triggers bug #44.

Applying fuzzing techniques to BtrPlace

From the analysis done in the previous subsections, in our fuzzer we should 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,
- the similarity of bugs so that our test-cases invoke different ones.

In order to achieve that, we can use the state-of-the-art techniques illustrated in the previous section, to create the core of the algorithm for the improved fuzzer:

1) Get the set of constraints and the interface of the scheduler.
2) Create a few tests for all the constraints
   - covering as most of the possible inputs as possible, taking into account all the different possible configurations,
   - using the transition probabilities as calculated in the previous sections,
   - exploiting a basic reducer that distinguishes test-cases with identical and similar inputs and remove them.
3) Create more test-cases for regions that prove to hide more bugs, thus direct the test-case production in an automated way (generate more inputs and further examine the input regions in which bugs are detected).
4) Create a ranked list of bugs, where those that are considered as more important should be higher in the list. In our case, crashes should be ranked higher, as they force the termination of the scheduler, followed by false-positive and then false-negative bugs.

The general sequence of bugs in the above algorithm resembles the Directed automated random testing approach.
The second step in particular implements the swarm testing approach, by trying to check all the possible different regions so that we can also get the system behaviour for a more diverse set of inputs. At the same time, at this step we exploit the diversity of configuration, as well as the similarity of bugs.
The third step resembles the feedback-directed test generation, as we generate more test-cases for input regions that bugs are already detected.
Finally, the last step is implementing the taming fuzzers approach, sorting the bugs according to their importance.

6. Conclusion

The VM scheduler is the cornerstone of the good functionality of IaaS clouds. However, the implementations of VM schedulers are defective, despite extensive testing using hand-written checks and unit-testing. The same applies for BtrPlace, a research-oriented virtual machine scheduler.

This tempted us to examine this problem and propose a possible solution. In fact, our study in this document included the following:

- Classification of bugs in crashes, false-negative and false-positive and extensive examination of representative bugs from the BtrPlace and the Nova schedulers. We made clear the consequences of each bug, provoking SLA violations and resulting to low Quality of Service.
- Proof that the current bug detection techniques in the BtrPlace scheduler are not sufficient. Even though unit-testing provides 80% code coverage and there are created 1000 lines of code for hand-written checkers, there are still reported bugs. In addition, the current fuzzer uses random actions on nodes and VMs, state transitions and constraint arguments, providing therefore low code-coverage, while some bugs are contained multiple times in the final results.
- Examination of state-of-the-art bug detecting and result optimization techniques. Fuzz-testing techniques, like directed-automated random testing, swarm testing and feedback-directed automated test generation proved to be quite appropriate for our solution. In addition, result optimization techniques like taming fuzzers and test-case reduction were also very interesting.
- Statement of the importance for an improved fuzzer for BtrPlace. We can achieve this by exploiting as many different scheduling configurations as possible and the calculating the probabilities for VM and node state transitions in a more efficient way. In the end, we proposed a new algorithm, based on the above principles and state-of-the-art techniques.

But our work on this topic does not stop at this document. In the following, we are going to implement the algorithm that we proposed in this document. Finally, we hope that we will succeed to verify the BtrPlace scheduler and generalize our work for all VM schedulers.

7. Bibliography

[1] Mell, Peter, and Tim Grance. "The NIST definition of cloud computing." (2011).
[2] Patel, Pankesh, Ajith H. Ranabahu, and Amit P. Sheth. "Service level agreement in cloud computing." (2009).
[3] 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.
[4] OpenStack Nova: http://nova.openstack.org/
[5] Nova open bugs: https://bugs.launchpad.net/nova/+bugs?field.tag=scheduler
[6] BtrPlace bugs: https://github.com/btrplace/scheduler/issues
[7] https://github.com/btrplace/scheduler/issues/48
[8] 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.
[9] https://github.com/btrplace/scheduler/issues/43
[10] https://bugs.launchpad.net/nova/+bug/1012822
[11] https://bugs.launchpad.net/nova/+bug/1227925
[12] https://github.com/btrplace/scheduler/issues/18
[13] https://github.com/btrplace/scheduler/issues/12
[14] https://github.com/btrplace/scheduler/issues/44
[15] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.
[16} Pacheco, Carlos, et al. "Feedback-directed random test generation." *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007.
[17] Whitebox testing: http://en.wikipedia.org/wiki/White-box_testing
[18] Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin. "Grammar-based whitebox fuzzing." *ACM Sigplan Notices*. Vol. 43. No. 6. ACM, 2008.
[19] Groce, Alex, et al. "Swarm testing." *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.
[20] Yang, Xuejun, et al. "Finding and understanding bugs in C compilers." *ACM SIGPLAN Notices*. Vol. 46. No. 6. ACM, 2011.
[21] McKeeman, William M. "Differential testing for software." *Digital Technical Journal* 10.1 (1998): 100-107.
[22] Chen, Yang, et al. "Taming compiler fuzzers." *ACM SIGPLAN Notices*. Vol. 48. No. 6. ACM, 2013.
[23] Zeller, Andreas, and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." *Software Engineering, IEEE Transactions on* 28.2 (2002): 183-200.
[24] Misherghi, Ghassan, and Zhendong Su. "HDD: hierarchical delta debugging." *Proceedings of the 28th international conference on Software engineering*. ACM, 2006.
[25] Agrawal, Hiralal, Richard A. DeMillo, and Eugene H. Spafford. "Debugging with dynamic slicing and backtracking." *Software: Practice and Experience* 23.6 (1993): 589-616.