Internship
2014 – 2015

[R|E]

D3
Thesis Report

Analysing bugs in VM schedulers

Participant:

- Alexandros TSANTILAS

Supervisors

- Fabien HERMENIER
- Ludovic HENRIO

Université Nice Sophia Antipolis

POLYTECH NICE-SOPHIA

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. We also present currently open bugs in the BtrPlace scheduler, dividing them according to an appropriate classification.
- secondly, we underline the difficulties in testing VM schedulers and we state some widely used techniques for testing software, including fuzzing approaches. In addition, we describe how the current verification tool of the BtrPlace works, underlining its weaknesses.
- then, we elaborate on our solution, that exploits the various configuration plans effectively to discover more bugs and includes a visualization tool that helps distinguishing similar and very different bugs.
- finally, we present our results and illustrate the causes of the bugs discovered by our tools. We then conclude that our solution is indeed effective, as we can identify more bugs and we can better distinguish their causes.

Table of Contents

1. Context & framework

1.1. 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.

1.2. 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, such as:

- 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.

1.3. 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 migrate 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 does not 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, while providers are subject to penalties when the SLAs are not met in practice.

1.4. 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.

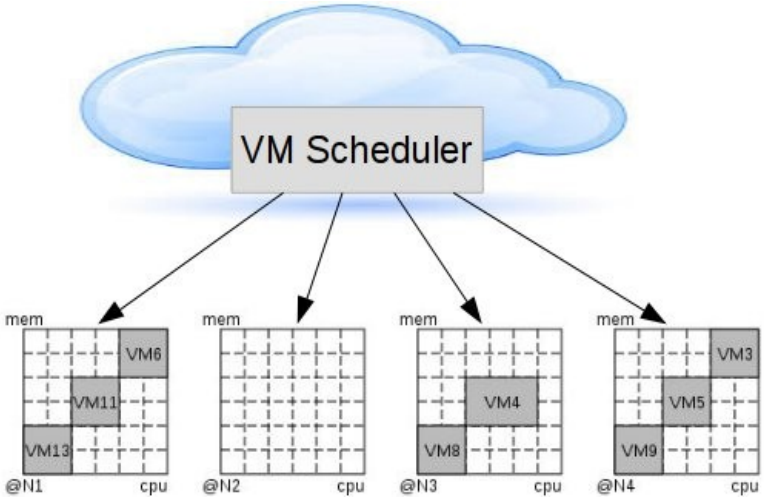

Figure 1: VM Scheduling example in BtrPlace.

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:

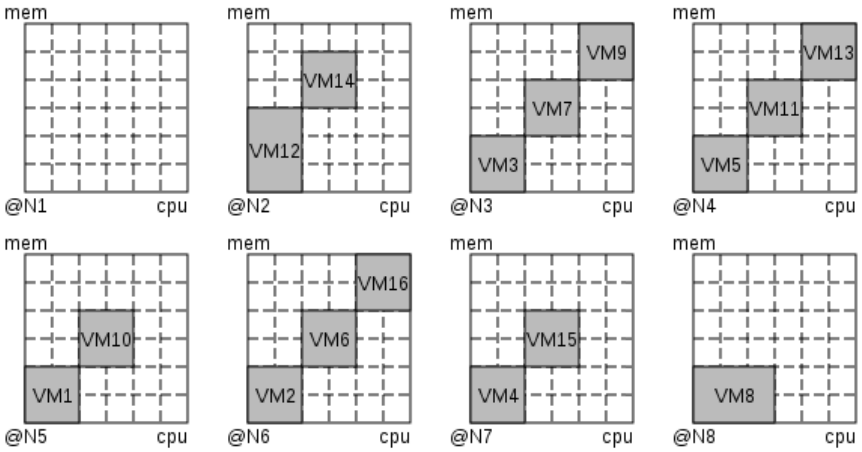

Figure 2: VMs initial configuration example.

Now, let's assume that we apply the following constraints. Their purpose is 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:

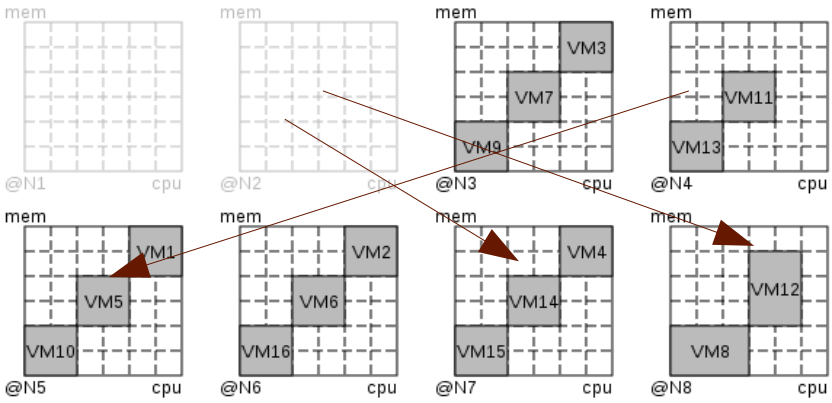

Figure 3: VMs final configuration after applying the constraints.

Another reconfiguration example in the BtrPlace can be the following. Initially, we have three VMs that are placed in the three physical nodes, while VM2 is in sleeping state, as shown in the following figure:

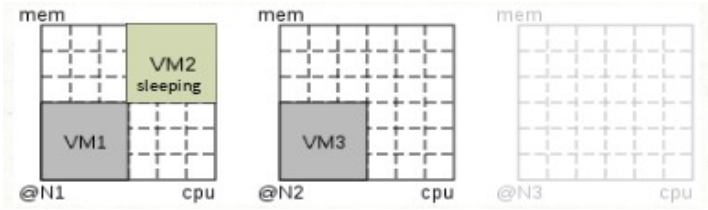

Figure 4: VMs initial configuration example.

Now, let's assume that we apply the following constraint, which demands that N1 should shutdown:

```
offline(@N1)
```

After the reconfiguration process, we would expect both VM1 and VM2 to migrate to a different node, as N1 goes offline. However, as we can see below, the reconfiguration actions do not include the migration of the sleeping VM2:

Figure 5: Schedule of the reconfiguration actions.

In the following figure, we can see the reconfiguration of the VMs:

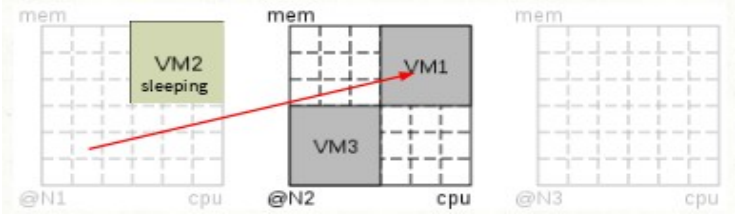

Figure 6: VMs final configuration after applying the constraints.

1.5. Bugs in VM schedulers

The last reconfiguration example of the previous section proves that there are bugs in the BtrPlace scheduler [6]. Similar bugs have been also observed in Nova, the component embedding the VM scheduler of the leading open source IaaS software stack OpenStack [4], where 16 bugs are still open about correctness issues [5].

In the following subsections we examine the three main categories of bugs found in VM schedulers. We elaborate on some representative bugs of the BtrPlace scheduler, to which we will often refer in the future, as well as the Nova scheduler. What is really important to consider at this phase is the difficulty in observing and provoking each category of bugs.

1.5.1. Bug categorization

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 VMs that are not in running state. 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 7: 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 test-cases that lead 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 8: 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 allowes 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 as well. 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.

For instance, the crashes 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.

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 in the following figure, 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 9: 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.

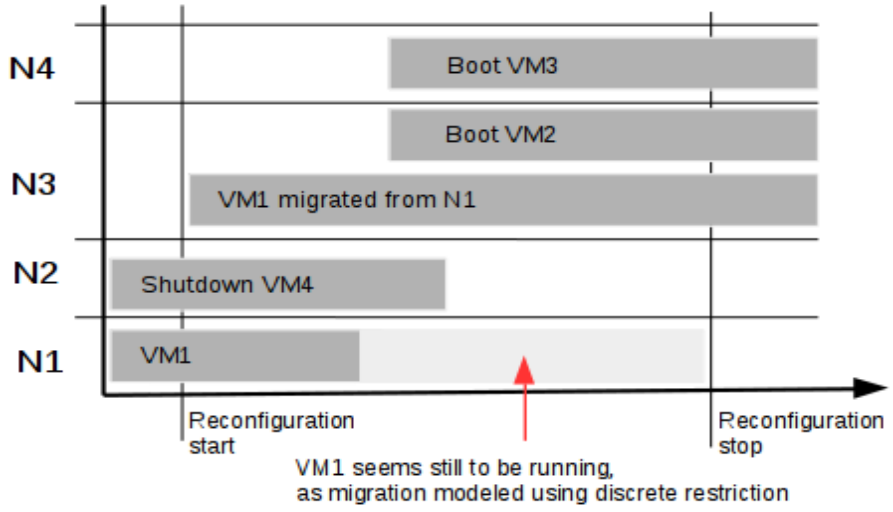

Figure 10: Demonstration of Bug #12.

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.

1.5.2. Consequences

As we can understand from the bugs observed, 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 and the development of incorrect schedulers with severe consequences for both clients and providers.

Bugs in VM schedulers can result to silent SLA violations, 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 particular, crashes (#48) that lead to runtime failures and false-positive bugs (#25, #43. #1227925) that lead to crashing reconfigurations, lead to bad Quality of Service (QoS) and SLA violations, as the requirement of the clients are not met. This can be translated to less income from clients.

False-negative bugs (#12, #18, #44) lead to resource fragmentation, which can be translated to loss of hosting capacity for the providers and finally results to less revenue for them as well.

2. Problem: How to test schedulers?

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 unidentified by the current bug detection tools.

2.1. Difficulties in testing VM schedulers

However, testing a VM scheduler is not an easy task as we have to consider the large number of different possible configurations and find a way to distinguish the bugs observed from each other. In the following subsections, we are examining the difficulties that are inherent in testing a VM scheduler.

2.1.1. Diversity of configurations

There are quite a few ways of producing different configurations. These include the transitions of the VMs and nodes, the possible time schedules of the actions and the constraint parameters and restrictions. In particular, let's assume that we want to test the research-oriented BtrPlace scheduler for the “Spread” constraint. We can calculate the number of all the possible reconfigurations as following:

1. The possible node states are two, online and offline. Therefore, the possible transitions are 4 in total:
   - offline to online or remain offline.
   - online to offline or remain online.
2. The possible VM source states are three, ready, running and sleeping. The destination states include the same as well as the killed stated. For this reason, the possible VM transitions are 12 in total:
   - ready to running, sleeping, killed or remain at the ready state.
   - running to ready, sleeping, killed or remain at the running state.
   - sleeping to ready, running, killed or remain at the sleeping state.
3. 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 5 different schedules (or less, depending on how exhaustive analysis we want to 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 simultaneously with t_1, for instance both t_1 and t_2 in [0:3].
   - 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].
4. We can have different constraint parameters. For the constraint “spread” and one VM and one node, we can have 3 different constraint parameters:
   - spread({vm1}),
   - spread({vm2}),
   - spread({vm1, vm2}).
5. Lastly, we can have both discrete & continuous constraint restrictions.

If we now sum-up all the different combinations we can have:

- 4 node transitions
- 12 VM transitions
- 5 time schedules
- 3 constraint parameters
- 2 constraint restrictions

In this case we have $4 \cdot 12 \cdot 5 \cdot 3 \cdot 2 = 1440$ different configurations for just 1 VM, 1 node and constraint “Spread”.

If we have 2 VMs and 2 nodes, we can't compare the relevant scheduling between the actions and therefore this parameter is omitted. The number of configurations in this case is calculated as:

- 4 node transitions for each of the 2 nodes,
- 12 VM transitions for each of the 2 VMs,
- $2^3 - 1 = 7$ constraint parameters,
- discrete and continuous parameter restrictions.

In total, we have $16 \cdot 144 \cdot 7 \cdot 2 = 32256$ different configurations to test.

Therefore, we can observe that there are numerous different configurations to test and that their number increases exponentially with the number of VMs and nodes. This fact makes very difficult to test a VM scheduler and cover all the possible configurations.

2.1.2. Identification of duplicate bugs

According to the differences that two inputs can have between each other, they can be denoted as similar, different and strongly-different. However, an effective testing technique should be able to distinguish if two bugs are a duplicate, no matter if they are triggered by similar or different configurations and therefore be able to report a distinct number of different bugs.

Below, we elaborate on each of the three categories and illustrate this categorization by providing example inputs that have the following three dimensions: $\begin{Bmatrix} VM\ transition \\ node\ transition \\ scheduling \end{Bmatrix}$

Similar inputs:

Such inputs usually trigger the same bug. This means that the bugs triggered by two similar inputs are in fact a duplicate and should be reduced by a test-case reducer. For example, let's consider the following two inputs:

$\begin{Bmatrix} running \rightarrow sleeping \\ online \rightarrow offline \\ vmAction < nodeAction \end{Bmatrix}$ and $\begin{Bmatrix} remain\ sleeping \\ online \rightarrow offline \\ vmAction < nodeAction \end{Bmatrix}$

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. However, the VM transition is not the same, but the destination state of the VMs is in both cases sleeping.
- The state transition for the VMs is happening before the state transition of the nodes.

These two similar 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 reducer of the testing framework that is used, should understand the similarity of the bugs and therefore merge them into one.

Different inputs:
Such inputs can trigger different but also the same bug, even though one or more (but not all) inputs are different. For example, let's have the following inputs:

$$\begin{Bmatrix} running \rightarrow sleeping \\ online \rightarrow offline \\ vmAction < nodeAction \end{Bmatrix} \text{ and } \begin{Bmatrix} sleeping \rightarrow ready \\ online \rightarrow offline \\ vmAction > nodeAction \end{Bmatrix}$$

These two inputs are going to trigger again the same bug, even though both the VMs transition and the scheduling are different. In this case, the reducer should again distinguish the similarity of the bugs, even though this case is more difficult.

However, we can also have the following inputs. In this example, the second configuration does not trigger any bug.

$$\begin{Bmatrix} running \rightarrow sleeping \\ online \rightarrow offline \\ vmAction < nodeAction \end{Bmatrix} \text{ and } \begin{Bmatrix} running \rightarrow sleeping \\ remains\ online \\ vmAction < nodeAction \end{Bmatrix}$$

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

$$\begin{Bmatrix} running \rightarrow sleeping \\ online \rightarrow offline \\ vmAction < nodeAction \end{Bmatrix} \text{ and } \begin{Bmatrix} vm\ migrates \\ remains\ online \\ overlapping \end{Bmatrix}$$

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

2.2. State-of-the-art

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, while unit testing consists 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, as we need to reveal as many bugs as possible, more extensive testing is required. As unit-testing is not efficient if there are lots of different possible inputs, fuzzing has become a more and more widespread testing technique to check complex software. In the following sections, we describe these techniques, as well as their implementation for the BtrPlace scheduler.

2.2.1. Unit testing

A unit is the smallest testable part of a system that can be invoked by a public interface. In object-oriented programming, a unit is often a single class, multiple classes working together to achieve one single logical purpose or even an individual method.

Unit testing is a software development process in which each unit is tested individually and independently for proper operation. The unit tests are created by programmers or by white box testers during the development process, in order to check the behaviour of every single unit. Each test case is independent from the others and this isolation can be often achieved using method stubs, mock objects, fakes and test harnesses. It consistently returns the same result, as the programmer always runs the same test, in order to test a single logical concept in the system.

However, unit testing is not able to catch every error in the program, since it is not possible to test a unit for every input scenario that will occur when the program is run. It can also be time-consuming and tedious in order to achieve the desired code coverage. In addition, it performs tests for the functionality of the units themselves. Therefore, it cannot detect integration errors and can't test functions performed across multiple units.

Unit testing in BtrPlace

In the BtrPlace there have been created eighty unit tests, with a code coverage of 80% achieved and one thousand lines of code written for hand-written checkers. Here, we can see an example of a unit test used to check the correctness of BtrPlace for the constraint “Spread”:

```
@Test
public void testEquals() {
    Model mo = new DefaultModel();
    Set<VM> x = new HashSet<>(Arrays.asList(mo.newVM(), mo.newVM()));
    Spread s = new Spread(x);

    Assert.assertTrue(s.equals(s));
    Assert.assertTrue(new Spread(x).equals(s));
    Assert.assertEquals(s.hashCode(), new Spread(x).hashCode());
    Assert.assertNotEquals(
            s.hashCode(),new Spread(new HashSet<VM>()).hashCode());
    x = new HashSet<>(Arrays.asList(mo.newVM()));
    Assert.assertFalse(new Spread(x).equals(s));
    Assert.assertFalse(new Spread(x, false).equals(new Spread(x, true)));
    Assert.assertNotSame(
            new Spread(x, false).hashCode(), new Spread(x, true).hashCode());
}
```

Figure 11: Unit test for the constraint “Spread”.

However, it is impossible to create a unit test for every possible configuration and every constraint for the scheduler, as we explained earlier that the number of configurations is extremely big. For this reason, it is necessary to examine other techniques for verifying BtrPlace and VM schedulers in general.

2.2.2. Fuzzing techniques

Fuzzing 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.

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.

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.

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.

2.3. Fuzzing in BtrPlace

As we examined earlier, unit testing cannot provide the desirable code coverage and is not able to examine a lot of different configurations for BtrPlace. For this reason, an automated bug detection tool that is using fuzz testing is also used currently.

2.3.1. WorkFlow

The current fuzzer's workflow is the following:

1. The fuzzer creates the configurations,
   - the probabilities for state transitions are static and predefined,
   - only the allowed transitions from the scheduler are created,
2. a reconfiguration plan is created for each test-case,
3. the constraint to test is compared to an invariant written using a domain-specific language, eg for the constraint “noVMsOnOfflineNodes” we have:

```
nodeState(n) /= online --> card(hosted(n)) = 0
```

4. a simulator checks the output of the constraint against its invariant.

2.3.2. Static transition probabilities

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.

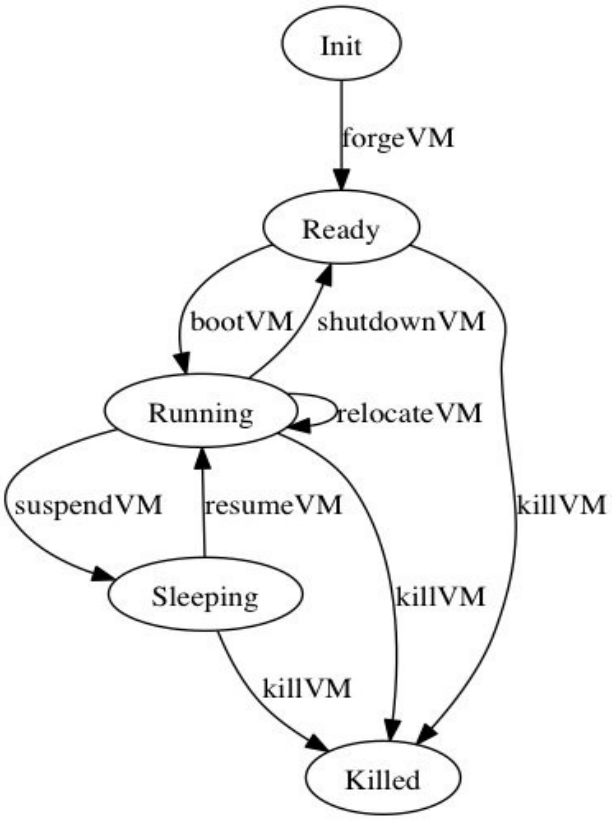

Figure 12: VM lifecycle in BtrPlace. There are five possible states that are changed on actions on Vms.

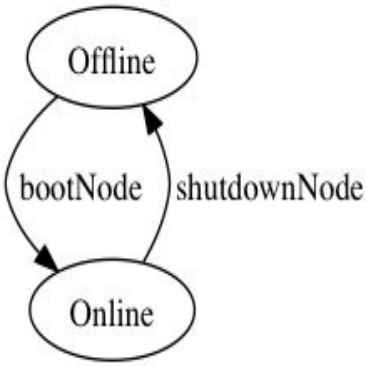

Figure 13: Lifecycle of a physical node.

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 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 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.

The current probabilities for the action transitions are hard-coded and very naïve. They 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.

As observed, the bugs at the “NoVMsOnOfflineNodes” constraint are triggered when a physical node is going from online to offline and there is a virtual machine in sleeping mode. If we consider that the VMs and the nodes have the same probability of being in any state and a “1 VM and 1 physical node” configuration that detects the bug, we have that:

- the initial state of the VM is sleeping or running with a probability of 2/3.
- 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:

$\frac{2}{3}.\frac{1}{2}.\frac{1}{4}.\frac{1}{2}=\frac{1}{24}$, so the possibility to find the bug is 1/24.

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); 3 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:

$\frac{2}{4}.\frac{1}{1}.\frac{1}{4}.\frac{1}{1}=\frac{1}{8}$, 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); 25 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].

2.3.3. Multiple occurrence of same 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.

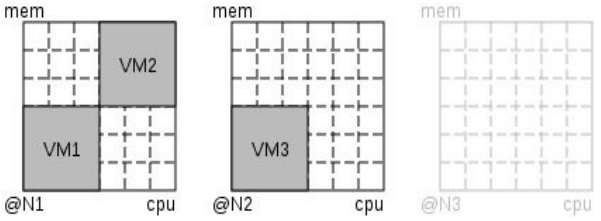

Figure 14: In this test-case if N2 with sleeping VM3 is allowed to shut-down, it is a bug.

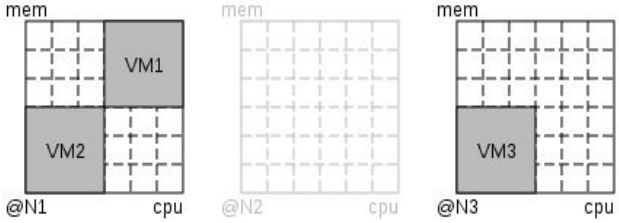

Figure 15: In this test-case, if N3 with sleeping VM3 is allowed to shut-down, It is bug. The two cases are similar.

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

As we can see in the results, it is the same bug. In the first case vm#1 on node#1 was in sleeping mode when the node was shutdown. In the second one it was exactly the same case, but it was vm#0 in sleeping mode in node#1. Another example can be the creation of two configurations with the same actions but different schedules between them.

2.4. Evaluation of current techniques

In the previous sections we analysed some state-of-the-art techniques for testing large software systems and their implementation in the BtrPlace testing framework. One of them is unit testing that verifies the behaviour of a very small testable component, by providing a predefined configuration and asserting that the outcome is the expected. This technique is proven insufficient for a VM scheduler due to the huge number of different configurations that can be applied.

The other technique is fuzzing, which can be a reliable solution for complex software systems. The current fuzzer has some useful features, like catching and distinguishing false-negative, false-positive bugs and crashes as well. However, it has also important weaknesses, such as low code coverage and identification of the same bug multiple times. This means that the test-case reduction is not sufficient, a technique whose goal is to construct a minimal test case that triggers the bug.

However, its current implementation in BtrPlace is not adequate and the current fuzzer has certain important weaknesses:

- it has low code coverage, as it is impossible to check all the possible reconfigurations of the scheduler,
- It creates configurations that identify the same bug,
- the test-case reduction applied is not very efficient as the same bug is reported more that once,
- the fault reports are not helpful for the developer and there is no visual tool for better result and bug representation.

3. Solution: Improved fuzzer for BtrPlace

Following the analysis of the previous chapter, it becomes evident that the current fuzzer should be improved. For this reason, we have to confront the limitations imposed by the current fuzzer and 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. The main challenges to overcome are:

- maximizing the code coverage of the fuzzer, making use of more efficient and more intelligent bug exploration techniques,
- creating few test-case scenarios that can identify the maximum number of bugs, instead of reporting numerous failure scenarios that hide the root causes,
- identifying distinct bugs and understand the similar ones.
- providing fault reports expressed in a way that assists the developer. This can be achieved by creating a visualization tool that demonstrates the configurations selected and helps distinguishing the different bugs from each other and finding their causes.

3.1. Workflow

In our proposed solution, the fuzzer no longer uses static transition probabilities and its goal is to create as many configurations as possible. According to the number of nodes and VMs that the configurations will contain, we implement the following rules:

- if we select configurations of 1 node and 1 VM, we examine all the different possible reconfigurations for each constraint, taking into account the VM and node transitions, the different schedules between the actions and the constraint parameters and restrictions.
- if we test 2 nodes and 2 VMs, we examine the scheduler for all the different VM and node transitions, leaving the schedules and the constraint parameters as random.
- if we test the scheduler more VMs and nodes, we keep the random behaviour of the current fuzzer but avoid the creation of similar configurations by keeping in a list the already examined configurations.

Finally, the basic workflow of the proposed fuzzer is the following:

1. the fuzzer creates all the reconfigurations that are going to be tested and a reconfiguration plan is created for each one,
2. the fuzzer selects the next configuration to be tested,
3. similarly with the current fuzzer, the constraint to test is compared to an invariant written using a DSL and
4. each plan is transformed in an appropriate expression, using a domain specific language
5. the results of each configuration are stored in a json file and
6. they are viewed in a different html page, using a diagram.

3.2. Description

In the next sections we describe the new features of the proposed fuzzer and the visualization tool that is used to present the bugs in an assistive way.

3.2.1. Range of tested configurations

The new fuzzer proposed can exploit all the different reconfigurations possible if we test the scheduler for one VM and one node. But as we saw before, we can have 960 different reconfigurations, just for the “Spread Constraint”.

If we want to test the scheduler *for just one VM and one node*, we consider for the proposed fuzzer three relative action schedules, being “overlapping”, “strictly before” and “strictly after”. However, we can compare a time schedule between two actions only if we have two real actions, as a VM or a node that is remaining at the same state is considered as a pseudo-action. In addition, a VM's initial state can be running or sleeping only if the node's state was initially online, else it can be only in ready state.

Therefore, our calculation changes as following:

- 2 node transitions (initial state is online) and 12 VM transitions, so 24 combinations.
  - Among these combinations, just 9 contain only real actions, so we can have 27 combinations.
  - The 15 remaining combinations cannot have a relative scheduling, so they are considered as overlapping.
- 2 node transitions (initial state is offline) and 4 VM transitions, so 8 combinations.
  - Among them, just 3 contain only real actions, so we can have 9 combinations.
  - The 5 remaining as considered as overlapping.

Therefore, we have in total $27+15+9+5=56$ combinations, which is the total number of different configurations of we consider a core constraint like “NoVMsOnOfflineNodes”.
This test takes less than one second to be completed, as we can see from the results below:

```
Bench.testNoVMsOnOfflineNodes: 56 test(s); 6 F/P; 2 F/N; 0 crash(s) (768ms)
```

If we test a non-core constraint like “Spread”, we also have to consider the parameters and the discrete/continuous restriction. In this case, for the constraint “Spread” we have:

- discrete or continuous restriction,
- $2^2-1=3$ constraint parameters: spread({vm1}), spread({vm2}), and spread({vm1, vm2}).

In total we have $56 \cdot 3 \cdot 2 = 336$ different configurations to test.

This test takes less than 2 seconds to be completed, as we can see below:

```
Bench.testSpreadContinuous: 168 test(s); 0 F/P; 0 F/N; 0 crash(s) (1017ms)
Bench.testSpreadDiscrete: 168 test(s); 0 F/P; 6 F/N; 0 crash(s) (322ms)
```

If we have *more than one VM and one node*, we consider random scheduling for the actions and random constraint parameters. In this case, if we have for instance 2 VMs and 2 nodes, we can calculate the number of configurations as following:

- we have 4 node transitions for each node, so 16 combinations in total,
- we have 12 VM transitions for each VM, so 144 combinations in total,
- we have $2^3-1=7$ constraint parameters,
- we have discrete and continuous parameter restrictions.

In total, we have $16 \cdot 144 \cdot 2 \cdot 7 = 32256$ different configurations to test.

This test takes around 40 seconds to be completed, but discovers bugs for the continuous constraint that were not discovered while testing the same constraint for just one VM and one node, as we can see below:

```
Bench.testSpreadContinuous: 16128 test(s); 203 F/P; 59 F/N; 0 crash(s) (22327ms)
Bench.testSpreadDiscrete: 16128 test(s); 0 F/P; 42 F/N; 0 crash(s) (18534ms)
```

If we do not consider any more the constraint parameters but just the restrictions and the VM and node state combinations, then we have $16 \cdot 144 \cdot 2 = 4608$ different configurations to test.

This test takes a bit more than 8 seconds to be completed, as we can see below:

```
Bench.testSpreadContinuous: 2304 test(s); 30 F/P; 4 F/N; 0 crash(s) (5062ms)
Bench.testSpreadDiscrete: 2304 test(s); 0 F/P; 5 F/N; 0 crash(s) (3332ms)
```

As we can see, with just two VMs and two nodes, the number of different configurations and the testing times become significantly greater, especially if we want to check all the constraint parameters and all the VM and node state combinations. For this reason, it is evident that for even more VMs and nodes we have to add even more randomness to our fuzzer. This can be done in three ways:

- totally random test-case creation, but storing of the already created tests so they are not tested again,
- generation of random but constraint-specific tests, eg:
  - for “NoVMsOnOfflineNodes” we can create only test-cases that include only VMs that their source or destination state is sleeping,
  - for “Spread” we can try to place more that one VMs at the same node,
  - for “Root”, we can try to migrate the Vms,
- creation of directed test-cases according to the bug patterns of the one VM and one node configuration results.

3.2.1. Bug visualization tool

As stated before, one of the most important elements of the proposed fuzzer is a bug analysis and bug visualization tool, with the help of which we can see the all of the tested configurations and understand the cause of the faulty situations. This visualization tool is in fact a sankey diagram representation of all the configurations tested.

A sankey diagram is specific type of flow diagram that is used to depict a flow from one set of values to another. The elements being connected are called

nodes and the connections are called links. Sankey diagrams visualize the flows and paths through a set of stages within a system.

In our case, every sankey diagram contains the source and destination states of the VMs and the nodes, the relevant scheduling between the actions and the constraint parameters. The description of each state in the diagram is defined using the Domain Specific Language used by the BtrPlace scheduler, while the bug category is defined by a specific colour, as shown in the next table:

Symbol	Meaning
^	source state
$	destination state
∀aV	for every VM action
∀aN	for every node action

Colour	Meaning
green	success
red	crash
orange	false-negative
dark red	false-positive

Figure 20: Diagram symbols and colors

Below, we can see the nodes of a sankey diagram and a successful test-case. The test-case configuration is the following:

- the node's source state is online and the destination state is offline,
- the VM's source and destination state is ready,
- the relative scheduling is overlapping,
- the constraint parameter selected is [vm#0, {node#0}] and the constraint restriction is continuous.

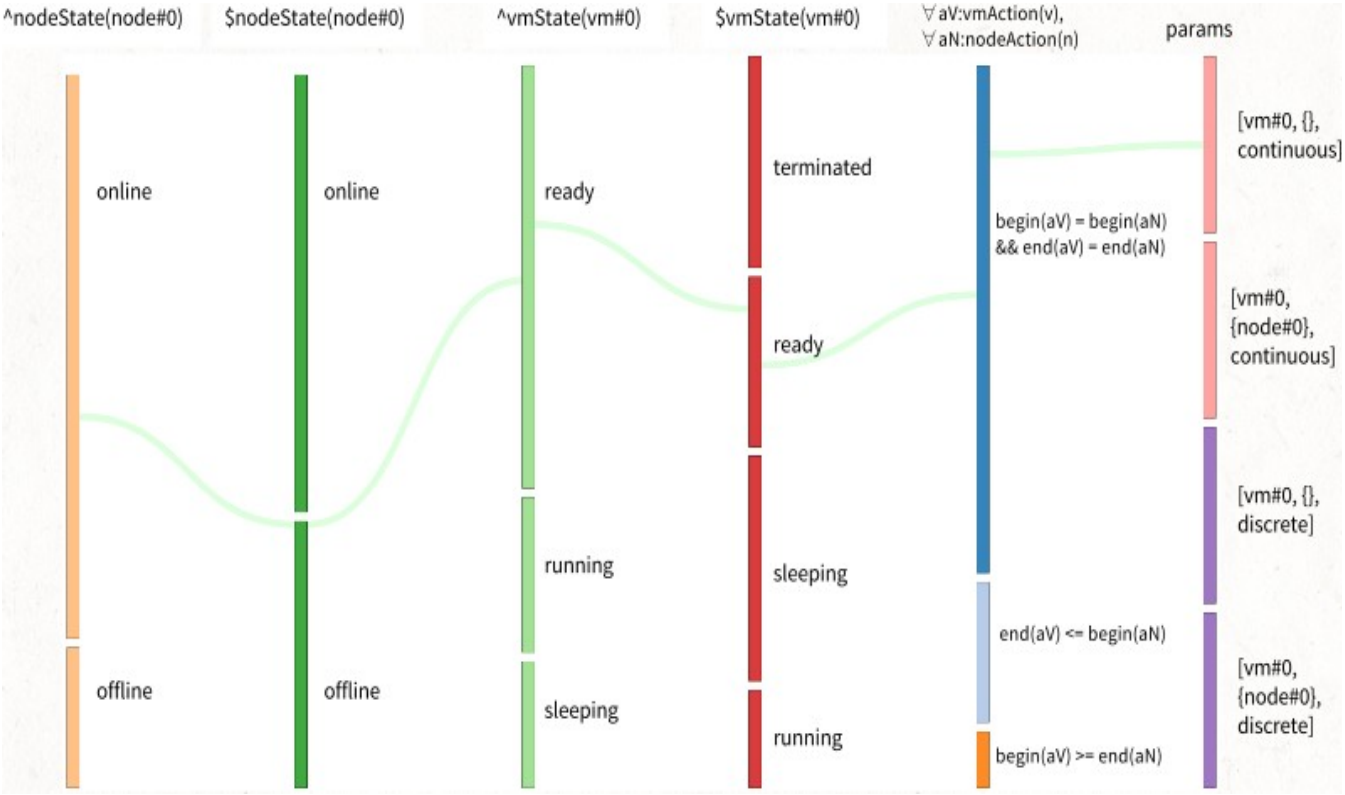

Figure 21: Sankey diagram representation for a successful test-case.

As mentioned earlier, the main purpose of the visualization tool is to present the faulty situations in a way that the developer can easily understand what are the root causes of the bugs. The general principle for discovering the cause of a bug is to find the common states between each of the faulty tests.

Below we can see the sankey diagram representation of the fuzzer results for the “Ban” continuous constraint, with configurations of one VM and one node. As we can see from the diagram, all of the false-positive tests pass from the sankey node that corresponds to the VM running source state and the node that corresponds to the [vm#0, {node#0}] parameters. In addition, we observe that there is faulty situation whatever the VM and the node destination state is, therefore the original issue is not the destination state. Thus, we can easily deduct that a false-positive bug occurs when we try to ban the running virtual machine vm#0 from the physical node node#0.

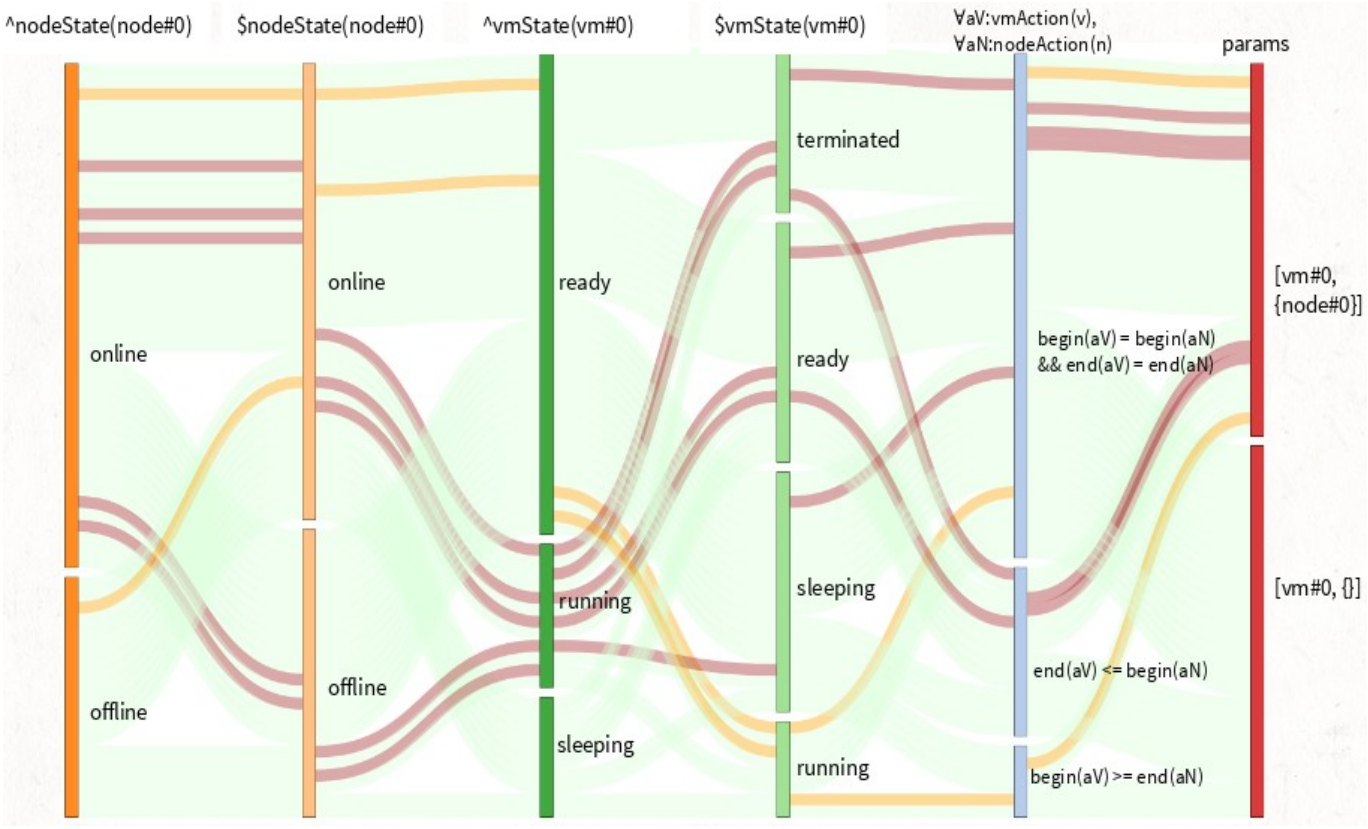

Figure 22: “Ban” continuous constraint

Another important feature of the bug visualization tool is that we can identify the similar from the different bugs. Again, if we realize that most of the faulty situations have a configuration element in common (eg a specific VM transition), then we can deduct that all bugs occur due to the same cause.

As we can observe in the following sankey diagram for the constraint “NoVMsOnOfflineNodes”, we have a total of six false-positive bugs. However, all of these faulty configurations contain some common steps:

- the node's source state is online and the destination state is offline,
- the VM's source or destination state is sleeping.

Therefore, we can deduct that there are not six different bugs, but it is the known bug “NoVMsOnOfflineNodes” that occurs when a VM is in sleeping state and the node hosting it is shutdown.

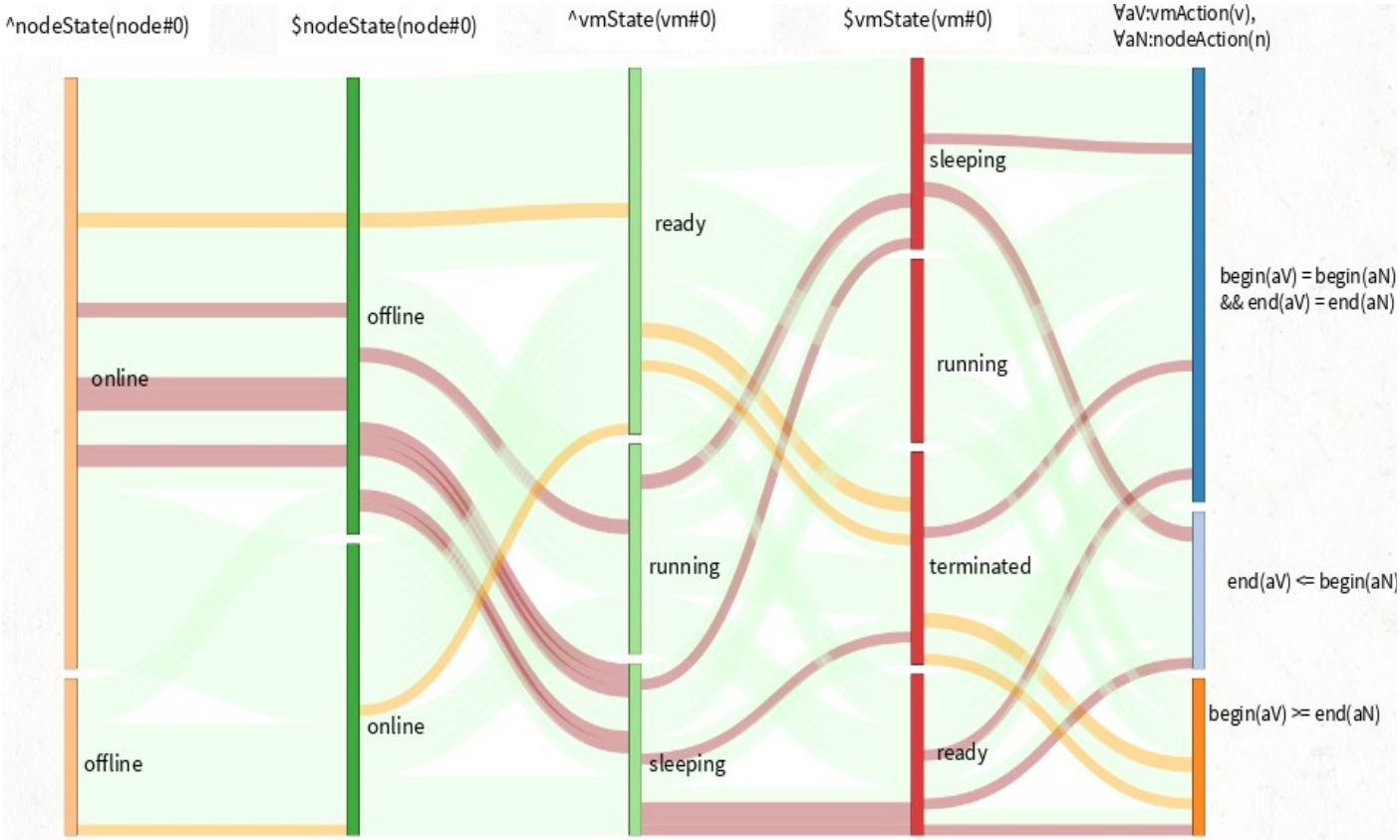

Figure 23: “NoVMsOnOfflineNodes” constraint

3.3. Implementation

For the fuzzer and the test-case creation, we wrote around 450 new lines of code and we also modified several existing classes. An important class is the “TestCasePlan” class. It contains all the necessary information for the configuration that is going to be checked, including the fields shown below, along with the required setters and getters, the constructor and implementation of the “equals” and “hashcode” methods.

```
private List<NodeStates> nodeSrc;
private List<NodeStates> nodeDst;
private List<VmStates> vmSrc;
private List<VmStates> vmDst;
private Schedule schedule;
private ReconfigurationPlan plan;
private List<Constant> args;
```

Figure 24: The fields of the TestCasePlan class

The core of the proposed new fuzzer is implemented in the “Fuzzer” class and is shown below. The “nodeSrcStates” collection is a list that contains all the possible initial combinations for the nodes. The same applies for “nodeDstStates”, “vmSrcStates” and “vmDstStates”. Another important classes that create the actions for the VMs and nodes according to their source and destination states are the “NodeStates”, “NodeActions”, “VMStates” and “VMActions” classes.

```
// Create the test-case recipe
for (List<NodeStates> nodeSrc : nodeSrcStates) {
   for (List<NodeStates> nodeDst : nodeDstStates) {
      // For vm source and destination states
      for (List<VmStates> vmSrc : vmSrcStatesFor(vmSrcStates, vmSrcStatesReady, nodeSrc)) {
         for (List<VmStates> vmDst : vmDstStates) {
            if ( (vmSrc.get(0) == VmStates.READY)
                  && (vmDst.get(0) == VmStates.TERMINATED)
                  && (nodeSrc.get(0) == NodeStates.online)
                  && (nodeDst.get(0) == NodeStates.offline) ) {
               System.out.println();
            }
            // For all the possible schedules
            // XXX : If we have 1vm : 1 node, we can have relative scheduling, else it is random
            Schedule[] schedules;
            if ( (nbNodes == 1) && (nbVMs == 1) ) {
               schedules =
                  Schedule.valuesFor(vmSrc.get(0), vmDst.get(0), nodeSrc.get(0), nodeDst.get(0));
            }
            else {
               schedules = new Schedule[]{Schedule.NO_SCHEDULE};
            }
            for (Schedule schedule : schedules) {
               TestCasePlan tcp = new TestCasePlan(schedule, nodeSrc, nodeDst, vmSrc, vmDst);
               ReconfigurationPlan plan = createReconfigurationPlan(tcp);
               ConstraintInputFuzzer cig =
                           new ConstraintInputFuzzer(cstr, new SpecModel(plan.getOrigin()));
               listParams = cig.createCombinations();

               // For all the constraint parameters
               for (List<Constant> params : listParams) {
                  possibilities.
                   add(new TestCasePlan(schedule, nodeSrc, nodeDst, vmSrc, vmDst, params, plan));
               }
            }
         }
      }
   }
}
```

Figure 25: The core of the proposed fuzzer

As for the visualization part, we initialize an html file for it and we create a json object that corresponds to it, as soon as a test campaign is starting to be executed. While the test campaign is running, all the test-cases created and run are added to the json object and when it is over, we flush the json object to the corresponding json file. The format of the json file is such that the d3.js library can parse it, obtain the data and present it as a web-page. It has two main elements, the sankey nodes and the links between them. The part of the sankey nodes section that describes the possible node states is the following:

```
{
 "nodes": [
   .............
   { "name": "^nodeState(node#0) \u003d online" },
   { "name": "^nodeState(node#0) \u003d offline" },
   { "name": "$nodeState(node#0) \u003d online" },
   { "name": "$nodeState(node#0) \u003d offline" },
  ...........
}
```

Figure 26: Nodes section for the possible node states

The links section that describes a full test-case configuration is shown below. Each “source” and “target” field denotes the start and the destination node of the link, as defined in the nodes section. If we examine the nodes section of the same json file, we realize that the following test-case is the:

- the node's source and destination state is online,
- the VM's source and destination state is ready,
- the relative scheduling is overlapping,
- the constraint parameter selected is [vm#0, {node#0}].


```
{
 .......
"links": [
 {
  "schedule": 1,
  "failures": "success",
  "vm0src": 11,
  "source": 4,
  "id": 62,
  "params": 15,
  "value": 1,
  "target": 6,
  "vm0dst": 11
 },
 {
  "failures": "success",
  "source": 6,
  "id": 62,
  "value": 1,
  "target": 8
 },
 {
  "failures": "success",
  "source": 8,
  "id": 62,
  "value": 1,
  "target": 11
 },
 {
  "failures": "success",
  "source": 11,
  "id": 62,
  "value": 1,
  "target": 1
 },
 {
  "failures": "success",
  "source": 1,
  "id": 62,
  "value": 1,
  "target": 15
 },
 .......
}
```


Figure 27: Links section for a test-case path

4. Results

4.1. Bugs cause analysis

In the following sections we present some bugs discovered by the proposed fuzzer, classified in the corresponding categories and then we evaluate the effectiveness of our fuzzer.

As a matter of simplicity, we analyse bugs that are discovered by configurations of one node and one VM. With such configurations we are able to test 13 out of a total of 17 constraints. In fact, we are not able to test the constraints “Among”, “MaxOnline”, “Split” and “SplitAmong”, due to the one VM and one node restriction for the configurations.

4.1.1. Continuous restriction not properly implemented

While testing various constraints, we found out that when selecting continuous restriction we had a lot of buggy situations. This means that it is not properly implemented and as a result we have the following consequences:

False-positive bugs

- in the continuous “offline” constraint all the configurations in which the node goes from online to offline.
- in the continuous “online” constraint all the configurations in which the node goes from offline to online.
- in the continuous “Ban” constraint the configurations in which the VM's source state is running and the parameters are [vm0, node0]. In this case, we try to ban vm0 from node0.

This problem was not suspected and with the proposed fuzzer it is confirmed. In fact, we found out that the source of the problem is that the initial state is not considered when having continuous restriction.

4.1.2. Missing transitions

We also observed that there is a bug when we have a transition to the terminated state. This bug is caused because the transitions to the terminated state were ignored and not implemented. In particular, we had the following faulty situations:

False-positive bugs

- the configurations in which the destination VM state is terminated, in discrete constraints “Sleeping” and “Ready”.
- the configurations in which the destination VM state is terminated and the node destination state is online, in discrete constraint “Running”.
- all the configurations in the continuous “Killed” constraint.
- the configurations containing running and terminated states for the VMs, in the discrete constraint “Killed”.

Crashes

- all other configurations in the discrete constraint “Killed”.

False-negative bugs

- in the discrete “Gather” constraint, the configurations in which the node goes from online to offline and then the VM goes from running to terminated.
- in the “NoVmsOnOffline” constraint, when the VM goes from ready to terminated, after the node action occurs.
- In the “Fence” constraint, when the VM goes from running to terminated.
- in the testing campaings “ToReady”, “toRunning”, “toKilled”, when the destination state of the VM is terminated and the transition of the VM is scheduled after the transition of the node.

This bug was also suspected before but was not observed by the previous fuzzer. However, with the proposed fuzzer there are lots of faulty situations in many constraints.

4.1.3. VM in multiple states

Another bug observed is that the VMs are allowed to be in multiple destination states. In particular, we observed the following faulty situations:

False-positive bugs

- in constraint “Quarantine” all configurations in which the VM's source state is running or sleeping and the destination state is running.

False-negative bugs

- in constraint “Quarantine” the configurations in which the VM's source state is ready and the destination is running.
- in testing campaigns “toReady”, “toRunning” and “toKilled” when the VM's destination state is running.

This bug was also suspected to exist in the BtrPlace, but was not observed from the previous fuzzer.

4.1.4. Action scheduling

Bug “Sleeping Vms on offline nodes”

As examined before, this bug appears when a node that is hosting a sleeping VM is going to the offline state. More, specifically, it appears when the following configurations are tested:

- in constraint “NoVMsOnOfflineNodes” when the destination state of the VM is sleeping, the node shuts-down and the VM action is scheduled before the action of the node.
- in constraint “NoVMsOnOfflineNodes” when the source state of the VM is sleeping, the node shuts-down and the two actions are overlapping.

It was a known bug and it was already observed by the previous fuzzer. This proves that our proposed fuzzer does not only discover new bugs, but finds the previously known as well.

<< Unidentified >>

I don't know if I should put them in section 4.1.3, because they occur a bit randomly, without following specific pattern and for sure not in section 4.1.1, as they are buggy for both continuous and discrete constraints:

RunningContinuous

whatever-running	FP

Crash when destVM != terminated, srcVM != running and destNode != offline

RunningDiscrete

FP when destVM != running and destNode != offline

ReadyContinuous

sleeping-whatever	Crash
whatever-running	FP
running-ready	FP

ReadyDiscrete

Same +

whatever-terminated	FP

SleepingContinuous

ready/sleeping-running	FP
running-ready/sleeping	FP
ready-sleeping/ready/terminated	Crash

SleepingDiscrete

whatever else contains ready/running/sleeping FP

ready-terminated	Crash

Lonely

Everything except for destVM = running Crash

4.2. Algorithm effectiveness

First of all, the proposed solution has a greater code coverage than the current fuzzer. This can be proved by the number of faulty situations it is able to discover. We are able to check the maximum number of different configurations, as no configuration is tested twice.

Secondly, the bug cause analysis we tried to figure out the root of the bugs that the new proposed fuzzer can identify. Among the bugs examined, there are already known bugs, like the “No VMs on offline nodes”, as well as bugs that were observed by the proposed solution. These include the unimplemented transition to the “Terminated” VM state, proving that our solution is able to observe not only the already discovered bugs, but also new.

Moreover, our algorithm is not slow as completing all the tests for one constraint and a configuration of 1 VM and 1 node needs less that 10 seconds. Similarly, testing a constraint having 2 VMs and 2 nodes requires around 40 seconds, which is negligible considering the code coverage achieved.

Lastly, the visualization tool provided really assists the developer in finding the cause of the bugs, categorizing them and distinguishing the similar from the different ones. Hence, the bug report is now more efficient and helpful.

5. Conclusion

5.1. Review

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.
- Analysis of the difficulties inherent in testing VM schedulers, like the numerous different scheduling configurations and the distinction of bugs, and 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, as well as result optimization techniques like taming fuzzers and test-case reduction prove to be quite interesting.
- 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.

- Presentation of our solution which consists on a directed way of producing test-cases in order to examine the most different configurations possible and a visualization tool, based on sankey diagrams, that can help us identify the bug causes and distinguish the various bugs from each other.
- Presentation of the results, that prove that our solution is efficient, being able to generate more distinct configurations and visualize the results in a better and more understandable way, that assists the developer in finding the cause and fixing the bugs.

5.2. Future work

Despite the work described here, there can be some important future improvements. The first main future goal would be to support more constraints, that can be achieved by allowing integers as parameters for the “counting” constraints. Secondly, it would be interesting if we could direct the transition probabilities when we have configurations of more than 3 nodes and 3 virtual machines, so that we have more chances of finding a bug. In the future, it would be intended to find and distinguish the bugs, by identifying and comparing their bug formulas. Then, by merging the bug formulas, we would have better source recognition and easier understanding of the bug causes.

6. 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.