Trustable Virtual Machine Scheduling in a Cloud

Fabien Hermenier
Nutanix
fabien.hermenier@nutanix.com

Ludovic Henrio
Université Côte d'Azur, CNRS, I3S, France
ludovic.henrio@cnrs.fr

ABSTRACT

In an Infrastructure As A Service (IaaS) cloud, the scheduler deploys VMs to servers according to service level objectives (SLOs). Clients and service providers must both trust the infrastructure. In particular they must be sure that the VM scheduler takes decisions that are consistent with its advertised behaviour. The difficulties to master every theoretical and practical aspects of a VM scheduler implementation leads however to faulty behaviours that break SLOs and reduce the provider revenues.

We present SafePlace, a specification and testing framework that exhibits inconsistencies in VM schedulers. SafePlace mixes a DSL to formalise scheduling decisions with fuzz testing to generate a large spectrum of test cases and automatically report implementation faults.

We evaluate SafePlace on the VM scheduler BtrPlace. Without any code modification, SafePlace allows to write test campaigns that are 3.83 times smaller than BtrPlace unit tests. SafePlace performs 200 tests per second, exhibited new non-trivial bugs, and outperforms the BtrPlace runtime assertion system.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software testing and debugging**; **Scheduling**;

KEYWORDS

VM scheduling, software testing, specification language

ACM Reference Format:
Fabien Hermenier and Ludovic Henrio. 2017. Trustable Virtual Machine Scheduling in a Cloud. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 12 pages.
https://doi.org/10.1145/3127479.3128608

1 INTRODUCTION

Infrastructure As A Service (IaaS) clouds provide clients with hardware via Virtual Machines (VMs). Inside the cloud, the VM scheduler is responsible for deploying the VMs to appropriate physical servers according to the established Service Level Objectives (SLOs). When environmental conditions (failures, load spikes, *etc.*) or the clients' expectations evolve, the VM scheduler reconfigures the deployment accordingly, using actions over the VMs and the servers. The SLO covers, for example, the expected availability, the minimum amount of resources to allocate, and possible placement constraints. Providers typically bill the client according to the amount of resources allocated and consumed, offset by any penalties to the provider incurred when the SLO is not met.

The VM scheduler is the cornerstone of the good functioning of an IaaS cloud. The provider bases his offering and the clients base their requirements on its features. The scheduler implementation is then expected to take decisions that are aligned with its documented behaviour. Implementing a VM scheduler that is *correct, i.e.* that behaves according to its documentation requires a strong domain specific expertise. For example, to implement a VM operation, a developer must understand the infrastructure management capabilities and ensure that the implementation matches the expected preconditions of this action. To implement a placement constraint, the developer must master several families of combinatorial problems (*e.g.* packing and task scheduling problems) and ensures that the code fits the many possible situations.

The difficulties to master every theoretical and practical aspects of VM scheduling leads to defective implementations that reduce the hosting capacity and thus reduces the provider revenue or breaks the SLO thus the client confidence. The current approach to develop VM schedulers and to ensure their quality is thus not effective. On the one hand, developers may not understand all the parameters that must be considered when they implement a component. On the other hand, common testing methods like unit testing, smoke tests, peer code-reviews or even static analysis are not sufficient to counteract the reasoning issues of developers.

In this paper, we present *SafePlace*, a testing framework to ease debugging of VM schedulers by checking that their implementation is correct against their expected behaviour. This solution is illustrated on BtrPlace [13], an open-source VM scheduler that received contributions from developers with various expertise. BtrPlace is also used in production by Nutanix, a provider of entreprise clusters, to mitigate local load spikes in thousands of private entreprise clouds[1].

The first contribution of this paper is *a Domain Specific Language (DSL)* to be used by the developers to specify the behaviour of VM schedulers. The second contribution is a complete software stack for *testing VM schedulers*. This includes first a parametric *fuzz testing* [20] environment to generate a large number of test cases and supersede the traditional unit tests that are long to write and biased by the developer perspectives. It also includes the *testing environment* to check the constraint implementations with regards to their expected behaviour. The third contribution is an analysis of the defects found in BtrPlace thanks to our technique. Our key results are:

Expressivity of the specification language. Constraint specifications are 50 characters long on average. The core of BtrPlace

[1]For more details, refer to http://www.nutanix.com

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3128608

was specified using 4 invariants. We specified 23 of its placement constraints that address affinity, resource allocation, state management, and hosting restrictions concerns. The specification language provides also the required features to specify all the constraints available in OpenStack Nova, VMWare DRS and the Acropolis Operating System (AOS) from Nutanix.

Usability of the testing suite. A test campaign is written with 3.83 times less lines of code than a single unit test of BtrPlace. A test campaign can run 200 different test cases per second which is fast enough to be executed directly from the development environment.

Effectiveness and identification of meaningful defects. SafePlace exhibited defects such as un-anticipated state transitions or action interleaving and aggressive optimisation that cannot be found using traditional testing methods. Those defects lead to crashes, SLO violations, and an under-utilisation of the resources. The specification based verification of SafePlace exhibit 2.23 times more defects than the assertion-based system of BtrPlace.

The rest of the paper is organised as follows. Section 2 discusses the internals of a VM scheduler and its possible bugs. Section 3 presents the specification language. Section 4 details the verification framework. Section 5 evaluates the benefits of the solution. Section 6 discusses the related works. Finally, Section 7 gives our conclusions.

2 BACKGROUND

A VM scheduler configures or reconfigures the allocation of the VMs to the different servers. Its solution must comply with the constraints provided by the user and the infrastructure. We first present in this section how VM schedulers compute such a solution. We then derive a defect taxonomy from an analysis of the issues reported in *OpenStack Nova*[2], *Apache CloudStack*[3], and *BtrPlace*[4].

2.1 The VM scheduling process

A VM scheduler computes suitable hosts for running the VMs depending on the state of the system, triggers resource allocation and schedule the actions to be overtaken. The *search space* associated to a problem to solve is then very large and the density of solutions varies depending on the workload and the infrastructure characteristics.

From a theoretical point of view, computing a solution consists of filtering the search space to only retain the decisions satisfying all the requirements. The filtering is explicit inside OpenStack Nova and CloudStack when the developers implement SLO enforcement algorithms. Each algorithm is a plugin that receives as arguments a set of possible hosts for the VM under control and removes among them those not satisfying the SLO specification. Each time the scheduler is invoked to decide where to place a VM, it chains the plugins to retrieve eventually the satisfying hosts and pick one among them. In BtrPlace, the filtering is implicit. The developer writes a constraint on top of a core Constraint Satisfaction Problem (CSP) to model the SLO. The specialised CSP is then solved using context specific filtering algorithms that prune from a search tree the branches where the constraint is not satisfied.

[2]http://www.openstack.org
[3]http://cloudstack.apache.org
[4]http://www.btrplace.org

2.2 Constraint taxonomy

Our analysis of the three schedulers exhibited 2 kind of constraints: core constraints and side constraints.

Core constraints are inherent to the hardware and the hypervisor capabilities. They restrict the reachable states for the VMs and the nodes depending on the current state. Core constraints specify the life-cycle of the elements under control and are then always enabled to ensure that the infrastructure is in a correct state. Core constraints prevent a VM to be booted if it is already running for example, or to live-migrate [6] a VM to an offline node. Core constraints also impose some precedence between actions. For example, by stating a running VM is necessarily placed on an online node, it prevents a node to go offline before all its hosted VMs are evacuated.

Side constraints allow the users to express their demand and implement a given SLO. A side constraint can have arguments to specify the elements under control or other SLO parameters. It can be guaranteed in a *discrete* or a *continuous* manner [7]. When discrete, the constraint must only be satisfied at the end of the VM (re)allocation process. When continuous, it must be satisfied at any moment of the element lifetime.

VM schedulers offer numerous side constraints to address a large range of concerns through different concepts. For example, the affinity and anti-affinity constraints consider the VM placement absolutely or relatively to other VMs or nodes. Resource matchmaking capabilities control the allocation of shareable resources (*e.g.* `Preserve` in BtrPlace) or enforce the presence of particular hardware on some nodes (*e.g.* `PciPassthroughFilter` in OpenStack Nova). Side constraints are also used to cap various counters. For example, they cap the number of VMs in a given state or location, or the number of online nodes (*e.g.* `NumInstancesFilter` in OpenStack Nova). Finally, some constraints address temporality issues (*e.g.* `NoDelay` to deny deferrable actions in BtrPlace).

2.3 Defects inside filtering algorithms

Inside OpenStack Nova, CloudStack and AOS, the code is checked with unit tests, modifications are peer-reviewed and smoke testing on a real testbed ensures that critical features are always working in practice. Finally, quality-assurance teams validate the features through practical stress testing on a testbed. The open-source code of BtrPlace constraints is checked by 81 unit tests, which provide 80% code coverage. The developer of a side constraint must also provide a checker to be invoked each time BtrPlace computes a schedule. In total, more than 4,000 lines of codes (22.7% of the codebase) are devoted to control the quality of BtrPlace side constraints.

Despite this state-of-the-art quality management for production systems, users still report critical defects due to invalid scheduling decisions. We derived 3 categories of domain-specific defects after an analysis of the public issue tracker of OpenStack Nova, Apache CloudStack, BtrPlace, and the private issue tracker of Nutanix related to its BtrPlace extensions.

A **crash** prevents starting VMs or reconfiguring placements. If the crashing conditions are deterministic and unrelated to a synchronisation of events, no scheduling is possible. Such defects were reported in BtrPlace [27], and CloudStack [2].

An **over-filtering** is a consequence of an algorithm that removes suitable solutions from the search space. Consequently, the scheduler ignores some viable solutions. Such defects were reported in OpenStack Nova [12, 14, 15], in CloudStack [23, 24], and in BtrPlace [9, 19].

An **under-filtering** is a consequence of an algorithm that does not remove unsuitable decisions from the search space. As a result, the retained scheduler decisions contradict at least one of the specified requirements. For example in OpenStack Nova, [5, 26] report concurrency issues that lead to server saturation, and thus low VM performance. In CloudStack, [25] allows the allocation of more than 100% of a storage unit to a VM. In BtrPlace, [8] shows that the implementation of the VM-VM anti-affinity is not consistent in some cases. In the AOS, their alternative implementation of VM-VM anti-affinity co-located some VMs under certain circumstances.

This study reveals that serious VM scheduler defects occur in situations that were not anticipated by the developers [7]. Crashes and under-filtering decrease the user confidence in the system. Over-filtering reduces the infrastructure hosting capacity and the return on investment for the IaaS provider. Such defects are also explained by a lack of expertise of the developers. Indeed, developing constraints requires a specific knowledge about the domain, its theoretical foundations (combinatorial optimisation), its logic (the VM and server lifecycle), and its pitfalls (over or under filtering decisions). Tracking defects requires a deep understanding of the possible states of a large infrastructure and the constraints it might face to infer an inconsistency.

3 DEBUGGING A VM SCHEDULER

This section discusses our approach to debug a VM scheduler. We first present a DSL dedicated to the specification of scheduling constraints, and then we illustrate its usage on the specification of some constraints of BtrPlace.

3.1 A DSL for specifying constraints

Numerous bugs are caused by a mismatch between the solutions that should be acceptable and the solutions that the scheduler algorithm considers. Developers should then be able to express formally what a valid solution is. Consequently, our DSL targets the specification of the *set of states accepted by a given constraint* and a natural approach is to rely on propositional logic. The *system state* can be defined as the set of elements that compose the system and the state of each element. Thus, specifications rely on set theory to reason on the state of the set of elements (VMs, nodes, *etc.*) and on dedicated functions to access the state of the elements. The set of dedicated functions is extensible. At the language level, a dedicated function is only characterised by a description and a signature. Its implementation must only be provided at runtime, using native code. This design makes the DSL adaptable to different business logics, and generic because a different set of elements and helper functions can be defined for each scheduler. Finally, the existence of temporal aspects inside constraints, especially in continuous constraints, leads to the introduction of a simple temporal operator to reason over the historical state of the elements.

Language design. The specification language is used both to express the core and the side constraints. It is based on first order logic augmented with a few simple set and list operators. Sets and lists can be defined both by extension and by comprehension but in practice only finite sets (and lists) can be defined; they may only be defined by either restriction of existing sets or by application of predefined functions. The language not only specifies acceptable configurations but also acceptable reconfigurations, where a *reconfiguration* is a set of operations allowing to go from a configuration to another, *e.g.*, migration, shutdown of a node, boot of a new VM. The generic syntax of the language is defined in Listing 1.

```
term ::=
    x // variable
  | term op term // operation
  | id ( term , .. , term ) // function call
  | ^id ( term , .. , term ) // temporal call
  | constant

  // set in comprehension or extension
  | { term . x typedef_op term, prop , .. , prop }
  | { term , .. , term }

  // list in comprehension or extension
  | [ term. x typedef_op term, prop , .. , prop ]
  | [ term , .. , term ]

typedef_op ::=  : | <<: | <: | /: | /<<: | /<:

op ::=
    + |  -  |  *  |   /  | /\  | \/
  | < | <=  |  >  |   >= |   =   | /=
  | & |  |  | --> | <->
  | : | <<: | <:  |  /:  | /<<: | /<:

prop ::=
    term  // boolean term
  | !x typedef_op term. prop
  | ?x typedef_op term. prop
```

Listing 1: Syntax of the constraint specification language.

Variables are either bound by the language constructs (by quantifiers and set definition), or predefined in the testing framework (e.g. `vms` is the set of all VMs). Identifiers `id` range over names of predefined functions and of other constraint specifications. `op` are basic operators on sets (intersection `/\`, membership `:`, inclusion `<:`, *etc.*), integers (comparison `=`,`<`, and arithmetic `+`,`-`, *etc.*), and boolean formulas (and `&`, or `|`, implication `-->`, *etc.*). The negation of those operators also exist, e.g. `/:` stands for $\notin$. `typedef_op` ranges over the operators allowed for variable definition, *i.e.* (finite) set membership, inclusion and packings. Packing being a special operator `<<:` such that `S<<:T` is true if `S` is a partition of a subset of `T`. Terms are additionally constrained by a type system that limits the expressiveness of the specification language so that it is simple, tractable, and intuitive. Calls can either be invocation of predefined functions or of constraint specifications. Predefined functions are simple functions defined in the testing framework. They consist of helper functions (e.g., *card*) and functions providing access to the system state (e.g., *host*, *vmstate*).

The *temporal call* construct is the most domain specific concept: it evaluates a function or constraint call, but giving to the variables

```
toRunning ::=
        !(v : vms) vmState(v) = running --> ^vmState(v) : {ready, running, sleeping}
toReady ::=
        !(v : vms) vmState(v) = ready --> ^vmState(v) : {ready, running}
toSleeping ::=
        !(v : vms) vmState(v) = sleeping --> ^vmState(v) : {running, sleeping}
noVMsOnOfflineNodes ::=
        !(n : nodes) nodeState(n) /= online --> card(hosted(n)) = 0
```

Listing 2: Specification of BtrPlace core constraints `toRunning`, `toSleeping`, and `toReady` restrict the possible state transition for the VMs. `noVMsOnOfflineNodes` forbids a node that is not online to host any VMs.

the values they had *at the beginning of the currently executed reconfiguration*. Temporal call enables reasoning on the historical state of the system without directly reasoning on time.

Comprehension sets are kept simple and easily computable: `x typedef_op term` defines a new variable ranging over a finite set defined by `typdef_op term`; this set is additionally filtered by checking for which value of `x` the propositions on the right are verified. Then the `term` is evaluated. The term can only use bound variables, *i.e.* `x` and the variables of the testing framework. Only finite sets can be generated this way; this is due to the constrained construction of comprehension set (variable bindings and typing) but also to the fact that the predefined functions cannot generate infinite sets when their input range over a finite set. The other constructs have a straightforward semantics.

Propositions are either terms (including first order logics) or quantified expression where the quantified variable is typed and restricted to a finite set, in a similar way as the comprehension set construct. A *constraint specification* associates a `formula` to an `id`. The behaviour of the core of the VM scheduler will thus be specified as a set of constraint specification, and each side constraint will be associated with a single constraint specification; our testing environment will check the conformance of a constraint with its specification.

A simple type-checker is implemented; it checks the type consistency of the parsed expression, e.g. that <: compares two sets of the same nature, and raises an error if a specification proposition cannot be typed.

Sample specifications. We illustrate below the semantics of our DSL based on the specification of some of the constraints of BtrPlace, OpenStack Nova, CloudStack, VMWare DRS and AOS. Listing 2 presents the complete specification of the core part of BtrPlace. This specification defines the lifecycle of VMs and nodes; it describes the state of the system and its possible evolutions. The first three specifications express the life cycle of the VM's state. For example, the first definition should be read *"for every VM, if the current state is running then at the beginning of the considered transition this VM should be ready, running, or sleeping"*. Remark that the specification uses several predefined functions. Furthermore, the ^ operator to reason on the transition of the system state (^*vmState* is a function that returns the historical state of the system). The last definition relates the node state and the VM state stating that *"for all nodes, if the node is not online then the number of VM hosted on this node should be 0"*. Matching the VM lifecycle of CloudStack would require removing the `sleeping` state while matching the OpenStack Nova lifecycle would require to add a `paused` state reachable from the `running` state.

```
RunningCapacity(ns <: nodes, nb : int) ::=
  sum({card(running(n)). n : ns}) <= nb

MaxOnline(ns <: nodes, nb : int)::=
  card({i. i:ns , nodeState(i)=online}) <= nb

Root(v : vms) ::=
  vmState(v) = running --> host(v) = ^host(v)

Lonely(vs <: vms) ::=
  !(i : vs) vmState(i) = running -->
                (hosted(host(i))) - {i}) <: vs

Fence(v : vms, ns <: nodes) ::=
  vmState(v) = running --> host(v) : ns

Split(part <<: vms) ::=
  {{host(v).v:p, vmState(v)=running}.p:part}
    <<: nodes

Among(vs <: vms, parts <<: nodes) ::=
  ?(g : parts)
     {host(i). i : vs, vmState(i) = running} <: g

SplitAmong(vs <<: vms, part <<: nodes} ::=
   (!(v : vs) Among(v, part)) & Split(vs)

MostlySpread(vs <: vms, nb : int) ::=
  card({host(v). v : vms, vmState(v) = running})
    >= nb

ShareableResource(id: string) ::=
  !(n : nodes)
    sum([cons(v, id). v : host(n)]) <= capa(n, id)
```

Listing 3: Sample specification of side constraints available in BtrPlace, VMWare DRS, OpenStack Nova, or AOS.

Listing 3 shows the specification of significant side constraints. `RunningCapacity` (`NumFilter` in OpenStack Nova) ensures that in total at most `nb` VMs are running on the online nodes among the set `ns`. It illustrates a basic set comprehension construct. `MaxOnline` ensures that at most `nb` nodes are online among `ns`. The specification illustrates the set comprehension construct that defines the subset of the nodes `ns` that are online. `Root` enforces a running VM that cannot be migrated, *i.e.* its next host always equals its current

host. `Lonely` (dedicated instances in EC2) states the host running the VMs in `vs` only host VMs in `vs`. When `Lonely` is coupled with the `Fence` constraint (`VM-Host` affinity in VMWare DRS), it equals `IsolatedHostsFilter` in OpenStack Nova. `Split` ensures that no two VMs belonging to two different sets are placed on the same node. This specification illustrates the packing construct. The constraint is specified in a positive manner that considers, for each set of VMs `p`, the set of nodes hosting running VMs of `p`; taking the different sets defined for each `p` in `part` should form a packing, thus ensuring that for two different `p`s their hosting nodes are disjoint. `Among` ensures a set of running VMs must be hosted on a single group of nodes among those available. This is specified by stating there exists one set `g` in the packing `parts` that is hosting every running VM in `vs`. With OpenStack Nova, such a constraint would be used for example to state a group of VMs must be running on a single availability zone. `SplitAmong` ensures distinct sets of running VMs must be hosted on a distinct set of nodes. With OpenStack Nova, such a constraint would be used for example to state every set of VMs are running on a distinct availability zone; providing high-availability if the VMs run a replicated service. Here, this constraint is specified using a composition of `Split` and `Among` inside different scopes. `MostlySpread` is the relaxation of the VM-VM anti-affinity constraint used in AOS. It ensures the given running VMs are hosted on at least a given number of distinct nodes. It supplants the common anti-affinity constraints available in OpenStack Nova, CloudStack, BtrPlace and VMWare DRS. `ShareableResource` is the constraint that prevent overbooking of a given resource on every node. *capa* and *cons* are business specific functions that return the resource capacity for a given node and the resource consumption for a given VM.

The examples in Listing 3 illustrate the adequacy of a propositional logic to specify SLA aspects of side-constraints. Furthermore, Listing 2 shows that the state transitions typical of core constraints can also be specified conveniently in our framework.

3.2 BtrPlace Specification

BtrPlace is written in Java. All the core constraints are embedded inside a single class that represents a prime filtering algorithm while each side constraint has a dedicated class. The formal specifications of BtrPlace constraints is done using annotations. Listing 4 illustrates the integration of the core constraint specification. Listing 5 illustrates the integration of a side constraint specifying the `Fence` constraint. For side constraints, the name of the constraint specification equals the class name while its arguments match the class constructor.

```
@CoreConstraint(name = "noVMsOnOfflineNodes",
                inv = "...")
@CoreConstraint(name = "toRunning", inv = "...")
@CoreConstraint(name = "toReady", inv = "...")
@CoreConstraint(name = "toSleeping", inv = "...")
public interface Scheduler {/*...*/}
```

Listing 4: Integration of the core constraints specification inside BtrPlace.

The use of annotations has two benefits. First, it integrates the specification as a part of the constraint documentation. It can be used as a substitute for the usual informal and ambiguous verbal specification or can complement it. Second, it is not intrusive as writing the specification does not modify any production code which is of a prime importance for developers.

```
@SideConstraint(args = {"v : vms", "ns <: nodes"},
  inv = "vmState(v) = running --> host(v) : ns")
public class Fence extends SimpleConstraint {

 public Fence(VM v, Collection<Node> ns) {/*...*/}
 /*...*/
}
```

Listing 5: Integration of the side constraint stating an anti-affinity between VMs and servers.

4 A DEBUGGING FRAMEWORK

The specification language allows to formalise the filtering process of a VM scheduler. In this section, we discuss how this specification is used to debug the implementation of a VM scheduler and BtrPlace in particular. We first detail the debugging methodology. We then discuss how test campaigns automatically generate test cases to reduce the developer effort.

4.1 Debugging Methodology

The exhaustive verification of a constraint implementation requires to explore the state-space of possible configurations and reconfigurations for the system. The state-space is however an unbounded space thus exploring it exhaustively is not possible. Accordingly, SafePlace uses a custom *fuzzer* [20] to pick randomly inside the state-space numerous *test cases*, each composed of an initial configuration, a reconfiguration plan, and constraint parameters. The reconfiguration plan consists of a set of actions, each with a determined starting and finishing time. It then checks for every picked test case whether the implementation behaves accordingly to the specification. The fuzzer only relies on a logical time to produce test cases where different interleaving of actions are investigated. In this context, the action duration is not meant for being realistic.

The constraint specification is the oracle that states the expected behaviour of the tested constraint. A specification evaluator is used to state whether a given system state is acceptable according to the specification or not. The state validation is performed on a representation of the infrastructure provided by a simulator that mimics the infrastructure manipulated by the VM scheduler and simulates the tested reconfiguration plan. The temporal calls are resolved by evaluating the functions of the initial configuration instead of the configuration currently reached. The test-case is analysed to check whether all the successive states reached during the reconfiguration plan are valid or not, according to the specification of the constraint under test. Each time an action starts or ends, the simulator updates the infrastructure state accordingly and tests the acceptability of the updated state. According to the semantics of BtrPlace, and of other existing VM schedulers, if several events occur at the same instant, then the termination of the actions finishing at this instant occur before the starting of new actions. The simulator applies the same scheduling and first checks the acceptability of the state reached after the finished actions, and then performs the starting actions

before checking again the state validity. Finally, the oracle *validates the test case* if and only if each state reached by the simulator is acceptable according to the specification evaluator.

To qualify the constraint implementation, SafePlace compares the output of the oracle against the VM scheduler output. It runs the VM scheduler in a constrained setting consisting of all the core constraints, plus optionally the side constraint to be tested. The VM scheduler is also constrained to only explore the reconfiguration planned in the test case. If it computes a solution, it is equivalent to the test case reconfiguration plan, this means that the test case is accepted by the constraint implementation. If the scheduler states there is no solution, then it means that the test case violates the constraint implementation. If both the oracle and the scheduler validate or invalidate the test case, then the implementation is consistent with the specification. If the scheduler validates the reconfiguration plan but the oracle does not, this means the constraint implementation under-filters. Finally, if the scheduler rejects the reconfiguration plan but the oracle validates it, this means the constraint implementation over-filters.

Testing conditions should be chosen precisely so that the diagnostic in case of inconsistency is unambiguous. Accordingly, SafePlace currently checks a single constraint at a time. As it explores all the possible reconfigurations, testing one constraint at a time is in general sufficient to ensure the correctness of the VM scheduler. Indeed, if a combination of several constraints causes a defect, it means that at least one constraint behaves differently from its specification and thus the testing of this constraint alone should identify the bug. Aside, testing the composition of constraints raises the problem of identifying the faulty one. Thus, when debugging a side filter, SafePlace constrains the test case so that it validates the core filter. When debugging a core filter, it constrains the test case so that it validates the other core filters.

Discussion. It is possible to misuse the DSL and have false positives or false negatives like with any specification based verification system. We think however that the use of a domain specific vocabulary limits this situation and such an approach has already been used in production in other contexts, *e.g.* at Amazon [22]. Furthermore, detecting mis-specifications is possible using peer review, for example, by asking developers if a given specification complies or not with a randomly generated test case, collect the answers and conciliate when they derive.

The approach is not exhaustive but the random picking already proved its effectiveness [28]. It also does not disrupt the developers' workflow thus ease the adoption of the methodology. The verification phase can be made online, directly on the developer's computer the same way he or she performs unit testing.

Finally, there is one last obvious restriction: the testing is driven by the modelling of the system and the bugs that happen independently of this model might not be found, *e.g.*, SafePlace may not detect that the VM scheduler might handle poorly machine failures if they are not considered in the model.

4.2 Debugging BtrPlace through test campaigns

Developers are not willing to change their development workflow. To ease the adoption of the testing methodology, the developer orchestrates *test campaigns* from the development environment or from the continuous integration system. Each test campaign generates test cases according to its fuzzer parametrisation and qualify constraint implementations accordingly.

A test campaign evaluates the implementation of a constraint in various conditions. For example, Listing 6 illustrates a minimal test campaign that focuses on the `lonely` constraint. As the constraint specification is attached to its Java implementation through annotations, the Java reflection API is sufficient to automatically generate the BtrPlace constraint from the test case. By default, the fuzzer generates reconfiguration plans having 3 nodes and 3 VMs, with an equal distribution between their initial and their destination state, a duration for each action varying between 1 and 10 seconds. The constraint arguments are generated randomly by picking values among the variables domain. Finally, the campaign runs up to 100 test cases and stops at the first failure. Despite a test over 3 nodes can be considered as small, Yuan *et. al* pointed out it is enough to reproduce the critical failures they observed in distributed data-intensive systems [29].

```
@CstrTest
public void testLonely(TestCampaign c) {
  c.fuzz().constraint("lonely");
}
```

Listing 6: A minimal test campaign.

The state space to explore to verify the constraint is unbounded. Even if we bound it, a long testing phase would alter the developer productivity. The developer can then customise the fuzzer using an internal DSL to make the test campaign focuses on a particular evaluation context. Because the specification language is extensible through business specific functions, the developer can also customise the fuzzer through decorators to make it add actions or events over the generated reconfiguration plan and instance.

```
1  @CstrTest
   public void testResourceCapacity(TestCampaign c) {

     c.fuzz().vms(10).srcVMs(0.1, 0.9, 0);

6    c.fuzz().with(
       new ShareableResourceFuzzer("cpu", 7, 10, 1, 5)
                                    .variability(0.5));

     c.fuzz().constraint("resourceCapacity")
11                       .with("id","cpu")
                         .with("qty",1, 50);

     c.fuzz().save("rc_test.json");
   }
```

Listing 7: Sample customisation of the fuzzer.

Listing 7 depicts such a customised test campaign. Line 4 tells to generate reconfiguration plans having 4 VMs with a probability of being initially in the `ready`, `running`, and `sleeping` state equals to 0.1, 0.9 and 0 respectively. The number of nodes and the state transitions will be unchanged. Lines 6-10 state to decorate the reconfiguration plans with a `ShareableResourceFuzzer`. By default, the fuzzed reconfiguration is only composed of actions related to

the state management and the VM placement. Such decorators allow to augment the generated plan to make it suitable to test some business specific constraints. `ShareableResourceFuzzer` generates random resource capacity and consumption for the nodes and the VMs. Here, it generates a `cpu` resource with a capacity varying between 7 and 10 for each node, a consumption varying between 1 and 5 for each VM. The plugin sets to 0.5 the probability to have for each VM a resource demand different from its current consumption. Lines 10-12 tells to test the constraint `resourceCapacity` with the string argument `id` set to `"cpu"`, to pick values between 5 and 10 for the integer argument `qty`. Finally, line 14 saves the generated test cases inside the file `rc_test.json`. This permits to replay a test campaign by re-using the test cases generated previously.

The developer also customises the test campaign to configure the scheduler, to declare the testing limits or to organise the tests. Listing 8 presents a test campaign customised for the sake of debugging the `fence` constraint. Line 1 states that the campaign belongs to the `affinity` group of test campaigns. Line 5 customises the scheduler. The developer enables the `repair` mode of BtrPlace to make it focus only on the VMs that are supposed to be misplaced [13]. Lines 6-7 illustrate a stopping condition stating that the test campaign must run up to 1000 tests, and stop at the first failure or after 10 seconds.

```
1 @CstrTest(groups={"affinity"})
  public void testFence(TestCampaign c) {
      c.fuzz().constraint("fence");

5     c.schedulerParams().doRepair(true);
      c.limits().tests(1000).failures(1)
                            .seconds(10);
  }
```

Listing 8: Sample test campaign customisation.

5 EVALUATION

This Section evaluates the benefits of SafePlace. We first evaluate inside BtrPlace its impact for the developers through a qualitative analysis of the specification language and the testing environment. We then experiment its ability to exhibit defects. All the experiments were executed on a Macbook Pro having one Intel I5 CPU at 2.9 GHz and 16GB RAM running OS X 10.12.3. Tests rely on the version 1.6.0 of BtrPlace. Experiments were launched directly from a development environment to reproduce the usage scenario of a developer testing his code.

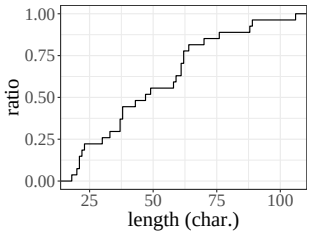
5.1 SafePlace integration

SafePlace is integrated inside BtrPlace through the specification language and the testing environment. This first experiment evaluates the integration by measuring the amount of code to write by a developer to specify and test a constraint.

BtrPlace specification. This evaluation estimates the effort for a developer to specify BtrPlace constraints using invariants. As the functions of the specification language are written in native Java code, the developer might also have to write new functions.

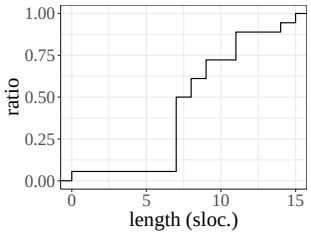
We specified 27 constraints of BtrPlace to validate the usability of the specification language. They formalise the states transition, the action scheduling, the resource sharing, the affinities between single element or groups of elements and the counting restrictions. We also specify `mostlySpread`, the side constraint inside the Nutanix Operating System presented in Listing 3. BtrPlace constraints are designed for being portable, they have an equivalent or they supersede all the affinity rules inside VMWare DRS and the filter scheduler of OpenStack Nova[5]. Consequently, the specification language provides the needed abstractions to also cover the constraints available in these two VM schedulers.

Figure 1 depicts as a Cumulative Distribution Function (CDF), the size of the BtrPlace invariants. Figure 2 depicts as a CDF the size of the 16 developed functions. In both cases, the measurement is made on a properly formatted code without using minification techniques.



Percentiles	Invariant length (sloc.)
50^{th}	47
95^{th}	88.7
100^{th}	106

Figure 1: CDF of the invariants length.



Percentiles	Function length (sloc.)
50^{th}	7
95^{th}	14.15
100^{th}	15

Figure 2: CDF of the functions length.

We observe that both the invariants and the functions are short. The median size of an invariant is 47 characters long and 95% of them are less than 89 characters long. The median size of a function is 7.5 sloc. long. The biggest being 15 sloc. long. It is crucial to have short and simple functions to be sure that they cannot create undesired behaviour like creating infinite sets or triggering an infinite loop. This conciseness confirms that a specification language based on first order logic, and domain specific functions is appropriate to specify the constraints while providing certain guarantees such as strong typing or the impossibility to define infinite sets.

Test campaigns. A first experiment evaluates the expressivity of the test campaigns against the 63 unit tests of BtrPlace that are devoted to the side constraints. The 166 other unit tests are ignored as there are either not focusing on a single constraint or focusing

[5]OpenStack Nova exhibits the largest number of filters but most of them can be rewritten as a VM-Host affinity constraint

other aspects of the scheduler. Figure 3 depicts the test length of SafePlace and BtrPlace as a CDF. We observe the code to write for the test campaigns are much smaller than for the unit tests of BtrPlace. The tests are 3.83 times shorter on average, with a median length of 6 sloc. for SafePlace against 23 sloc. for BtrPlace. This conciseness is desirable for a developer as it contributes to increase in the proportion of time spent on writing production code. This expressivity is explained by the features provided by the test campaigns. First, the fuzzer relieves the developer from writing specific test scenarios that can naturally introduce a bias. Second, the expression evaluator relieves the developer from evaluating the adequacy of the result. Thus, within a test campaign, the developer only specifies if needed, the campaign parameters to make it focus on a particular context.

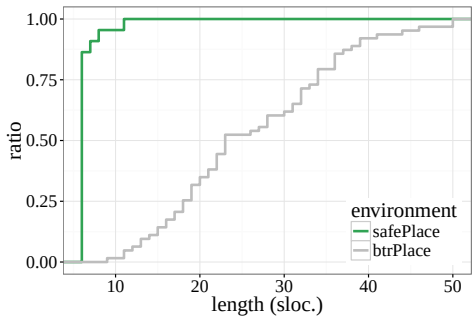
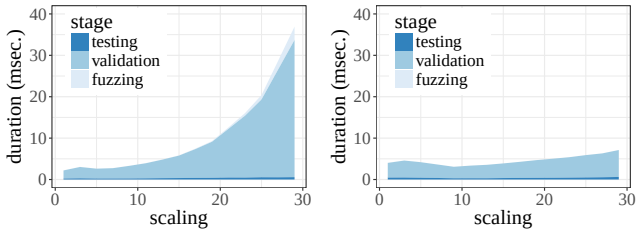


Figure 3: Test length.

The second experiment evaluates the testing speed of SafePlace. For this experiment, each test campaign runs 100 test cases, the number of VMs and nodes vary from 1 to 30, and the possible initial states or reconfigurations. Figure 4 shows the average test duration and its distribution. In Figure 4a, any initial state and reconfiguration are allowed for the VMs and the nodes. It is then possible to generate test cases that contradict the core constraints. In Figure 4b, the fuzzer is tuned to prevent any violation of the core constraints.



(a) Fuzzing over the complete VM and node life-cycle.

(b) Fuzzing with online nodes and initially running VMs.

Figure 4: Testing duration and its distribution. The fuzzing stage generates a test case. The validation stage validates the test case against the core constraints and re-generate test cases upon violation. The testing stage evaluates a valid test case.

We first observe that the validation stage dominates the testing phase. This is explained by the 4 restrictive core constraints to check during the validation stage while 1 constraint is evaluated during the testing stage. We also observe that the duration increases with the instance size. The core constraints require to test every VM and node. As a result, the duration increases at least linearly with the number of elements. Finally, we observe that the duration varies depending on the allowed states and transitions Indeed, when fuzzing over all the possible states and transitions, the risk of having at least one core constraint violation increases exponentially with the instance size (see Figure 4a). As discussed in Section 4, the validation stage is required to ensure no constraint other than the tested one can fail. In the case the developers are confident with the core constraint implementations, it might then be acceptable to modify the fuzzer so that it generates instances that necessarily validate these constraints. Accordingly, the validation stage could be removed from the testing workflow.

The unit tests of BtrPlace usually focus on instances made up with up to 5 VMs and nodes which is considered sufficient to exhibit most of the failures that are reported inside representative distributed systems [29]. This leads to test cases that run quickly and exhibit faults that are easy to fix due to the limited number of artefacts. With a test campaign that generates instances having 10 VMs and 10 nodes, it is already possible to test the implementation against numerous combinations of element states and transitions, subsets and packings. At this scale, SafePlace runs 200 tests per second. This allows developers to keep testing the production code interactively within their IDE. Test campaigns that are generating big instances or tens of thousand of instances might slow down the developer productivity if they are launch within their IDE. In that case, the test campaigns should be executed as a part of the continuous integration environment.

5.2 SafePlace benefits

This experiment evaluates the capacity of SafePlace at finding defects inside BtrPlace. A test campaign is executed for each of the 22 side constraints and runs 1,000 test cases. At total, 22,000 test cases are then generated, executed and analysed. All the reported defects were confirmed by BtrPlace developers.

Table 5a summarises the defects, classified by the consequences for the end user, the number of involved constraints and the number of failing test cases. SafePlace finds defects in 12 of the 22 evaluated constraints, all being side constraints. Under certain circumstances 10 of the constraints under-filter, leading to decisions that violate the SLOs; 3 constraints crash the scheduler; and 6 constraints over-filter and reduce the infrastructure hosting capacity. We also observe that the under-filtering is the most common consequence (57.02% of the defects).

Table 5b summarises the defects by their underlying cause. This taxonomy is derived from a manual analysis of the defects to infer a generic cause. Our analysis, under the supervision of BtrPlace developers, confirms that our approach does not exhibit false positives. Consequently, it also confirms the defects are not caused by a mis-specification or an error in the simulator. The cause named *A* is the biggest source of defects. It refers to the use of a continuous constraint in a context where the constraint is initially violated. In

Consequence	Constraints	Failing tests
Under-filtering	10	938
Crashes	3	459
Over-filtering	6	244

(a) Defect consequences.

Code	Defect cause	Constraints	Tests
A	initial violation in continuous restriction mode	7	704
B	unexpected arguments	4	642
C	discrete filtering in continuous restriction mode	3	45
D	unsupported action synchronisation	4	20
E	bad action semantic comprehension	1	16
F	unconsidered initial element state	1	4

(b) Defect causes.

Figure 5: Cause and consequences of the defects and their distribution.

past releases of BtrPlace, some constraints automatically switched to the discrete mode in that situation. To prevent inconsistencies, the implementation must now report the violation. This signals that the developers forgot to fix the old behaviour in some corner cases. The cause *B* refers to implementations that are not robust enough to support valid but unusual arguments. Those are usually empty collections or packings with some empty subsets. This reveals that the developers are over-confident with the parameter content. The cause *C* refers to an aggressive optimisation of the code. The developer uses an optimisation that is fair in the discrete mode but inadequate in the continuous mode. This reveals that the developers neglect the temporal dimension of the scheduling problem and only consider the spatial dimension. The cause *D* refers to an unanticipated interleaving of actions such as an ignored precedence relationship between two actions. This reveals that the developers cannot always consider all the possible action interleaving. The cause *E* reveals a misunderstanding of the action semantic. Here, the developers ignore that despite a migrating VM is hosted simultaneously on its source and its destination, it is still running solely its source host. This reveals that developers lack knowledge about the action model. Finally, the cause *F* highlights the situation where the developers omit the complete life-cycle of the elements.

There is currently 3 open issues in BtrPlace that are related to filtering problems. 12 corresponds to cause *E*, 44 to cause *C* and 121 to cause *A*. All were reproduced and reported automatically by SafePlace.

This experiment validates the capacity of SafePlace to exhibit existing and new defects. More importantly, the analysis of their root causes confirms practically that i) the defects are specific to the domain of VM scheduling and ii) SafePlace eases their detection and their reporting.

Evaluating the expression evaluator. Since 2011, BtrPlace contains an assertion system to verify the validity of each computed reconfiguration plan. The developer of each constraint is expected to write a checker used inside a simulator. The checker API is event based and the developer can write code inside up to 24 methods. This task is error prone, not attractive enough for the developer and we observed that the last implemented constraints do not include a checker. SafePlace could effectively replace this checker as both tools automatically check the consistency of a solution. In this experiment, we then compare the detection capabilities of the checker and of SafePlace by running test campaigns using either the constraint specification or the checker as an oracle.

Figure 6 shows that the SafePlace methodology finds much more defects than the BtrPlace checkers: while the checkers reported 249 defects, the expression evaluator reported 556. In this experiment, we used the assertion-based checker of BtrPlace to check the configurations that were produced by the SafePlace fuzzer. This experiment shows two informations. First, the difference between the two checkers highlights that the traditional assertion-based checker of BtrPlace is unable to identify over-filtering defects. Second, the SafePlace fuzzer was also able to exhibit 249 scenarios that were not anticipated by the developer despite being meaningful. More qualitatively, we observed that a checker never reported a defect that was not also exhibited by SafePlace.

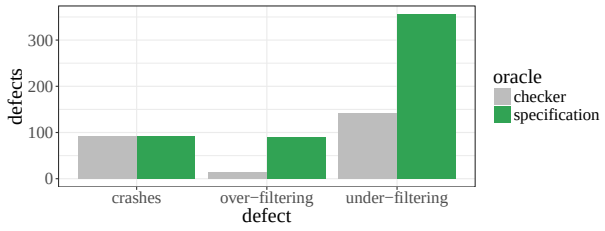


Figure 6: Defects exhibited using SafePlace or BtrPlace checkers.

All these experiments validate the benefits of using a specification language coupled with testing techniques to detect defects. SafePlace enabled to exhibit reasoning issues that are specific to the domain of VM scheduling. These defects cannot be captured through traditional code analysis tools such as those relying on static analysis, concolic testing or symbolic execution. The use of a specification language plays also a role of formal documentation while being faster to write than a checker.

5.3 Verifying BtrPlace advanced features

This last set of experiments revise some of the features of BtrPlace that modify the way constraints are applied. For these experiments, each test campaign generates and solves 500 instances having 10 VMs and 10 nodes each. By default, the fuzzer generates random test cases. To perform an accurate comparison, the test cases have been generated and stored to be reused in every context.

Continuous constraints in practice. BtrPlace supports discrete and continuous constraints [7]. A discrete constraint must be

satisfied at the end of a reconfiguration while a continuous constraint must be satisfied at any moment of the reconfiguration. This experiment compares the implementation quality of these restrictions. It runs the test campaign of the 9 constraints that support both a continuous and a discrete mode. Figure 7 shows the results and reveals that the continuous implementation of the constraints is less robust than their discrete implementation. 236 of the test cases exhibit an inconsistency in the discrete mode against 987 in the continuous mode. In details, while the continuous implementation over-filters slightly more than the discrete implementation (103 additional test cases), 854 of the test cases exhibited an under-filtering. The purpose of the continuous constraints is to enforce the SLO satisfaction at every instant. In this context, having a trustable scheduling algorithm is even more important than in the discrete case because the SLOs that must be ensured in a continuous way are the most critical ones. This experiment also confirms that it is more difficult for the developers to master the scheduling problems than the placement problems as implementing a correct continuous constraint is error prone.

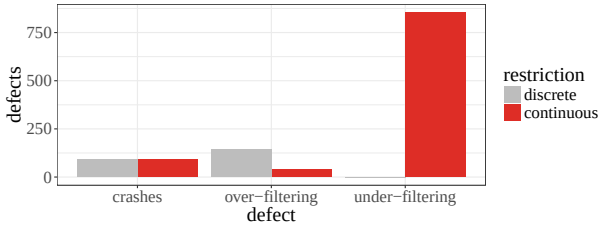


Figure 7: Defects depending on the restriction mode.

Problem simplification in practice. The repair mode of BtrPlace forces the filtering algorithm to focus only on the set of VMs that are possibly mis-placed, the other being untouched [13]. To compute this set, each constraint uses a heuristic to estimate, with regards to its filtering, the VMs it suspects. If the heuristic over-estimates this set, the solving duration of the problem increases. More importantly, if the heuristic under-estimates the set of VMs, then the scheduler might no longer be able to find a solution; this is a particular case of over-filtering.

This experiment measures the under-estimation risks of the repair mode. We run two test campaigns with and without the repair mode enabled and analysed the defects. SafePlace reports the same defects in both situations (578 over 10,000) (see Figure 8). Each time, the defects are present with or without the repair mode. The heuristics used inside each constraint do not lead to an over-filtering. Thus, this legitimates their use to increase the scheduler performance.

6 RELATED WORKS

To the best of our knowledge, SafePlace is the first approach to address correctness of VM schedulers. However, SafePlace combines formal verification and testing techniques to identify faults in cloud computing. Accordingly, this section discusses these three research domains that inspired this work.

Faults in cloud computing. Cloud systems are supposed to be always available. The research community is thus interested in the

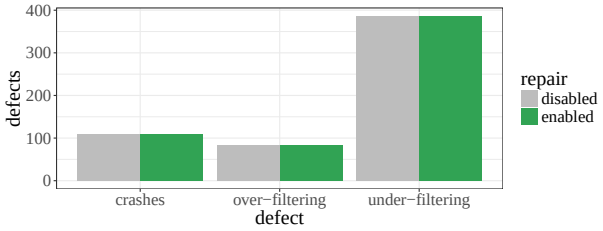


Figure 8: Defects depending on the repair mode status.

characterisation and correction of faults in cloud systems. Gunawi *et al.* analysed bugs in Hadoop Map-Reduce, ZooKeeper, HBase, HDFS, Cassandra and Flume [10] and then in cloud service outage [11]. They report that *logic specific* bugs are the most frequent, partially caused by a gap between the system specifications and the resulting code.

None of these works analyse VM schedulers. The focus on this critical component led us to a domain specific defect taxonomy, a specification language bringing implementation and specification together, and a testing environment to report domain specific reasoning issues.

Formal verification in Cloud Computing. Massively used for critical systems, formal methods start to be used by the cloud community. Naturally, a first use case being to validate critical components such as the hypervisor. For example, Leinenbach *et al.* [16] verified a subset of the Microsoft Hyper-V hypervisor instructions while Amit *et al.* [1] used the Intel Hardware validation tool, to perform a test-based validation of the virtual VCPU behaviour inside the KVM hypervisor. Amazon used TLA specifications coupled with the TLC model checker to exhibit new bugs inside Amazon S3 and DynamoDB [22]. These approaches provide the highest level of trust. They however require extensive skills in formal methods that prevent their massive adoption by the developers. This was witnessed by the Facebook engineers that were not used to the TLA+ concepts, and designed a C-like alternative, closer to their usual programming language. In our case, the temporal notion required in VM schedulers is less important than in those approach, this is why our DSL first rely on propositional logic where constraints can be specified clearly and briefly.

With SafePlace, we also provide to the developer a tractable approach. The use of a DSL eases the constraint specification while fuzz testing reports significative defects without requiring modifying the production code nor to change the developer workflow.

Automated testing of cloud systems software. Bouchenak *et al.* [3] test PaaS services from a specification based on a finite-state automaton. This approach is meaningful at the PaaS layer because the notion of workflow is inherent to such services. Concerning VM schedulers, a DSL based on propositional logic is more suitable to represent SLA constraints than a specification based on state transitions. Unit tests and smoke testing are the standard tools to detect bugs. They are effective when the developer knows what matters to test, and when the awaited results can be checked rigorously. However, Section 2 pointed out that tests are biased by the developer expertise. Yuan *et al.* also reported that despite rigorous testing protocols, numerous cloud systems are still failing due to

interleaving of events that remained untested despite being simple to reproduce [29]. Verification tools based on static or dynamic analysis can prevent from a biased testing. For example, Klee [4] mixes symbolic execution and a SAT solver to generate unit tests that can ensure 100% code coverage given enough time. Symbolic execution suffers however from scalability issues when the codebase is large or contains a complex control flow. An alternative is fuzz testing [20]. This method generates random input data for a component to detect crashes. For example, CSmith [28] detects compiler crashes on correct C source code. These solutions detect bugs with a small effort from the developers. However, they are usually limited to the detection of crashes and cannot exhibit reasoning issues as they cannot compare the execution output against the expected behaviour. SafePlace being a testing environment for VM schedulers, it reports along with crashes, over-filtering and under-filtering issues thanks to the constraint specification language.

Formal verification of process schedulers. Inside a multicore system, a process scheduler elects at regular interval the processes to run and assign them to physical cores. Accordingly, the process scheduler shares a notion of placement with the VM scheduler. Because a process scheduler is also a critical component, the research community proposed solutions to address performance and dependability concerns. For example, Bossa [21] proposes a DSL to ease the correct implementation of single-process schedulers. The DSL ensures, among other things, that the transitions for a process state are valid with regards to the specified automaton while a core never executes a blocked thread. After a recent study of defects inside the Completely Fair Scheduler of the GNU/Linux Kernel [18], Lepers *et al.* [17] start to enhance Bossa with the objective of generating a proved optimistic multicore scheduler. SafePlace targets correct (VM) schedulers and also uses a formal language to ease the verification process. however our approaches differ significantly. Lepers *et al.* use an imperative language being able to generate a correct scheduler code at a latter stage while SafePlace currently uses a declarative language to formalise the awaited state of the system and verify legacy VM schedulers.

7 CONCLUSION

VM schedulers do not always take decisions aligned with the advertised behaviour. These defects have consequences in terms of trusts and revenues. They are caused by the difficulty to master the theoretical and practical aspects of placement problems, and to envision all the situations that must be considered when implementing the algorithms that filter out the unsatisfactory decisions.

We proposed SafePlace to exhibit consistency issues in VM scheduler code. SafePlace mixes formal specification and fuzz testing to report inconsistencies between the scheduler specification and its implementation. The developer writes a specification of the constraints using a DSL based on first order logic and domain specific functions. The associated implementation is then automatically checked for conformance.

We validated SafePlace on the VM scheduler BtrPlace. The integration does not require changing the production code, nor the development workflow. The specification language is portable enough to formalise the constraints inside BtrPlace, OpenStack Nova, VMWare DRS and the Acropolis Operating System from Nutanix. Specification is concise, 50-character long per constraint. The testing code is 3.83 times smaller than BtrPlace unit tests and can generate and run 200 tests per second. SafePlace found all the currently open bugs related to the filtering algorithms. Furthermore, despite the current 80% code coverage of BtrPlace, SafePlace exhibited new high-level defects with 5 of the 6 root causes being reasoning issues specific to the domain of VM scheduling; undetectable by static analysis. Those defects led to crashes, SLO violations, and resource wasting. Finally, SafePlace outperformed the legacy assertion based checkers by revealing 2.23 times more defects.

AVAILABILITY

SafePlace is available as a module of BtrPlace under the terms of the LGPL v3 licence. It can be downloaded at http://www.btrplace.org.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Ennan Zhai for their feedback on the paper.

REFERENCES

[1] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 311–327. https://doi.org/10.1145/2815400.2815420

[2] Bharat Kumar. 2014. CloudStack – live migration is failing for vm deployed using dynamic compute offerings with NPE; unhandled exception executing api command: findHostsForMigration java.lang.NullPointerException. https://issues.apache.org/jira/browse/CLOUDSTACK-6099. (2014).

[3] Sara Bouchenak, Gregory Chockler, Hana Chockler, Gabriela Gheorghe, Nuno Santos, and Alexander Shraer. 2013. Verifying Cloud Services: Present and Future. *SIGOPS Oper. Syst. Rev.* 47, 2 (July 2013), 6–19. https://doi.org/10.1145/2506164.2506167

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[5] Chris Friesen. 2015. OpenStack Compute (nova) – race conditions with server group scheduler policies. https://bugs.launchpad.net/nova/+bug/1423648/. (2015).

[6] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 273–286. http://dl.acm.org/citation.cfm?id=1251203.1251223

[7] Huynh Tu Dang and Fabien Hermenier. 2013. Higher SLA Satisfaction in Datacenters with Continuous VM Placement Constraints. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep '13)*. ACM, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/2524224.2524226

[8] Fabien Hermenier. 2016. BtrPlace – continuous spread fail. https://github.com/btrplace/scheduler/issues/123. (2016).

[9] Fabien Hermenier. 2016. BtrPlace – CShareableResource and 0 capacity/consumption. https://github.com/btrplace/scheduler/issues/124. (2016).

[10] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2670979.2670986

[11] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/2987550.2987583

[12] Hans Lindgren. 2013. OpenStack Compute (nova) – Affinity filter checks erroneously includes deleted instances. https://bugs.launchpad.net/nova/+bug/1107156/. (2013).

[13] F. Hermenier, J. Lawall, and G. Muller. 2013. BtrPlace: A Flexible Consolidation Manager for Highly Available Applications. *IEEE Transactions on Dependable and Secure Computing* 10, 5 (Sept 2013), 273–286. https://doi.org/10.1109/TDSC.2013.5

[14] Jay Lee. 2014. OpenStack Compute (nova) – Disk filter should not filter for boot volume. https://bugs.launchpad.net/nova/+bug/1358566/. (2014).

[15] Jinquan Ni. 2015. OpenStack Compute (nova) – Affinity policy problems with migrate and live-migrate. https://bugs.launchpad.net/nova/+bug/1497100/. (2015).

[16] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods (FM '09)*. Springer-Verlag, Berlin, Heidelberg, 806–809. https://doi.org/10.1007/978-3-642-05089-3_51

[17] Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix, Redha Gouicem, Julien Sopena, Julia Lawall, and Gilles Muller. 2017. Towards Proving Optimistic Multicore Schedulers. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 18–23. https://doi.org/10.1145/3102980.3102984

[18] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326

[19] Michael Drogalis. 2016. BtrPlace – No solution is found. https://github.com/btrplace/scheduler/issues/112. (2016).

[20] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[21] Gilles Muller, Julia L. Lawall, and Hervé Duchesne. 2005. A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '05)*. IEEE Computer Society, Washington, DC, USA, 219–230. https://doi.org/10.1109/WORDS.2005.7

[22] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73. https://doi.org/10.1145/2699417

[23] Nitin Mehta. 2013. CloudStack – CLONE - Allocation capacity of a cluster during HA. https://issues.apache.org/jira/browse/CLOUDSTACK-4941. (2013).

[24] Prachi Damle. 2013. CloudStack – userconcentratedpod_firstfit failed to find alternate host to run VM. https://issues.apache.org/jira/browse/CLOUDSTACK-2158. (2013).

[25] Rajani Karuturi. 2015. CloudStack – Allocated percentage of storage can go beyond 100%. https://issues.apache.org/jira/browse/CLOUDSTACK-8896. (2015).

[26] Robert Collins. 2014. OpenStack Compute (nova) – gap between scheduler selection and claim causes spurious failures when the instance is the last one to fit. https://bugs.launchpad.net/nova/+bug/1341420/. (2014).

[27] Vincent Kherbache. 2014. BtrPlace – Out Of Bounds Exception. https://github.com/btrplace/scheduler/issues/48. (2014).

[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[29] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Symposium on Operating System Design & Implementation*. USENIX Association, Berkeley, CA, USA, 249–265. http://dl.acm.org/citation.cfm?id=2685048.2685068