Generating Test Sequences to Assess the Performance of Elastic Cloud-based Systems

Michel Albonico, Stefano Di Alesio, Jean-Marie Mottu, Sagar Sen, Gerson Sunyé

Generating Test Sequences to Assess the Performance of Elastic Cloud-based Systems

Michel Albonico[1,2], Stefano Di Alesio[3], Jean-Marie Mottu[1], Sagar Sen[3], and Gerson Sunyé[1]

[1]AtlanModels team (Inria, IMT-A, LS2N), France
Email: {jean-marie.mottu, gerson.sunye}@inria.fr

[2]Federal University of Technology - Paraná (UTFPR), Brazil
Email: michelalbonico@utfpr.edu.br

[3]Certus Centre for Software Verification & Validation, Simula Research Laboratory, Norway
Email: {stefano, sagar}@simula.no

***Abstract*—Elasticity is one of the main features of cloud-based systems (CBS), where elastic adaptations, such as those to deal with scaling in or scaling out of computational resources, help to meet performance requirements under varying workload. There is an industrial need to find configurations of elastic adaptations and workload that could lead to degradation of performance in a CBS, serving possibly millions of users. However, the potentially great number of such configurations poses a challenge: executing and verifying all of them on the cloud can be prohibitively expensive in both, time and cost. We present an approach to model elasticity adaptation due to workload changes as a *classification tree* model and consequently *generate short test sequences* of configurations that cover *all T-wise interactions* between parameters in the model. These test sequences, when executed, help us to assess the performance of elastic CBS. Using MongoDB as a case study, test sequences generated by our approach reveal several significant performance degradations.**

***Index Terms*—Cloud computing, Elasticity, Combinatorial testing, Performance testing.**

I. Introduction

In cloud computing, *elasticity* is the property of a system to provision (*scale-out*) and de-provision (*scale-in*) resources in response to workload variation [13]. The main goal of elasticity is to make the cloud-based system's (CBS) resources match the current workload as closely as possible, in order for the user to always experience an acceptable performance level. An elastic behavior is typically achieved through *adaptations*, i.e., dynamic changes in the structure and operational parameters of the CBS. For example, in Database Management Systems (DBMS), an elastic adaptation may consist in replicating data into new virtual machines to ensure that data access does not constitute a bottleneck [7]. Other scenarios where elasticity plays a key role in maintaining a steady performance level are load peaks (*flash crowds*), where adaptations typically consist in rapidly deploying stateless virtual machines of the CBS services.

Inadequate adaptations to resource provision can result in CBS performance degradations, and consequently an unsatisfactory service to the users. Therefore, it is important to assess the performance of CBS by testing their elasticity. A typical strategy to do so consists in varying CBS workload, and observing whether the elastic adaptations of the systems lead to significant degradation in performance. The number of interactions between elastic adaptations and possible workloads can be very large. In addition, these combinations (or *configurations*) should be executed in a sequence, mimicking a real-world scenario. However, testing all possible sequences (arranging configurations in distinct ways) can be prohibitive due to time associated with running each configuration on the cloud. Depending on the number of configurations, it could take weeks. Therefore, we must reduce the number of configurations in order to reduce execution time, while still being possible to find performance degradations.

In this paper, we present an approach based on Combinatorial Interaction Testing (CIT) to reduce the number of configurations in a satisfactory manner. The approach consists of three steps: (1) We model the domain of elastic adaptations and workload, i.e., the *elasticity parameters*, as a classification tree [3]. This helps us to define the bounds of a search space for test configurations. (2) We leverage on proven CIT algorithms [3], [16], [22] to generate a minimal set of configurations that covers all T-wise interactions between elasticity parameters. (3) We develop a graph traversal algorithm to generate valid test sequences of configurations, i.e., test sequences satisfying constraints that specify how adaptations can occur on the cloud.

We conduct a systematic experiment with test sequences that cover pairwise (2-wise) interactions between elasticity parameters. These test sequences help us to identify several performance degradations in MongoDB, a CBS case study, and to reveal performance degradations of distinct severity. Preliminary results, considering a larger coverage of elasticity parameters, suggest that pairwise reveals most of the elasticity-related performance degradations of our case study. However, we left a more complete investigation of larger coverage of elasticity parameters for future work.

The remainder of this paper is organized as follows. Section II introduces the principles of cloud computing and combinatorial testing. Section III presents our testing methodology. Section IV describes the experiments and we discusses their results. Section V presents the threats to validity. Section VI presents the related work. Finally, Section VII concludes and lists future perspectives.

II. Background

In this section, we describe the main aspects of cloud computing elasticity and combinatorial testing.

A. Cloud Computing Elasticity

Figure 1 presents an example of the typical behavior of elastic CBS. In particular, the graph reports the *resource demand* (y-axis) expressed over time (x-axis) as a percentage of the currently allocated resources. The demand linearly increases from 0 to 1.5, and then decreases to 0. Note that a resource demand equal to 1.5 means that the application demands 50 % more resources than the current allocated ones.

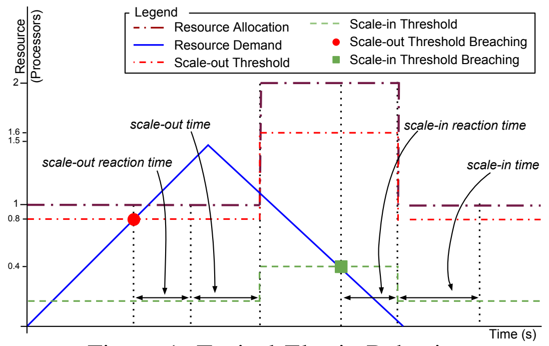

Figure 1: Typical Elastic Behavior

If the resource demand exceeds the *scale-out threshold* for the *scale-out reaction time*, the cloud elasticity controller instantiates a new resource, which becomes available after a *scale-out time*, i.e., the time the cloud infrastructure needs to allocate it. Once the resource is available, the scale-in and scale-out threshold values are updated accordingly. In a similar way, if the resource demand becomes lower than the *scale-in threshold* for the *scale-in reaction time*, the cloud elasticity controller releases a resource.

Elastic CBS' behavior can be translated into three elasticity-related states: *scaling-in*, *scaling-out*, and *ready*. Figure 2 depicts the transitions between these states.

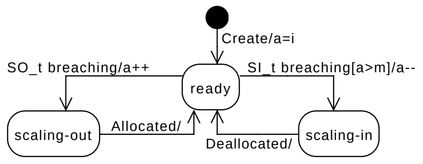

Figure 2: Elasticity States

At the beginning, the CBS is launched (event *create*) and enters into the *ready* state. In this state, no resources are being allocated or deallocated. Upon this initial transition, the number a of allocated resources is initialized to a given value i ($a = i$). If the cloud system is exposed to an increasing demand of resources that breaches the scale-out threshold (*SO_t breaching*, and the number of allocated resources is lower than the maximum available ($a < Max$), the cloud elasticity mechanism starts allocating a new resource (a++). At this point, the cloud system enters into the *scaling-out* state, and remains there until the resource is completely allocated. After a *scaling-out*, the cloud system returns to the *ready* state. When the demand of resources breaches the scale-in threshold (*SI_t breaching*), and the number of allocated resources is larger than the minimum provided ($a > min$), the cloud elasticity mechanism starts to *deallocate* a resource (a–). This leads the cloud system into the *scaling-in* state until the resource is deallocated, and finally the system returns to *ready* state.

B. Combinatorial Interaction Testing

In complex cloud computing systems, the elastic behavior is determined by a large number of parameters, such as workload thresholds, virtual machine type, and system topology. The interaction between some of these parameters may be the cause of system failures, or performance degradation at runtime. Exhaustively testing all possible combinations of parameter values, i.e., all possible *configurations*, is often computationally too expensive, because the total number of combinations of parameter values quickly increases as the system size grows.

Several techniques have been proposed over the years to address the intractability of exhaustive testing by selecting a smaller representative set of configurations. Among those, Combinatorial Interaction Testing (CIT) is a strategy that consists in testing all T-wise combinations of the parameters of a system [20]. This means that, when considering n parameter values, T-wise combinatorial testing investigates only $2^T \cdot \binom{n}{T}$ configurations compared to the $n!$ required for exhaustive testing. Kuhn et al. show that 2-wise (pairwise) coverage of parameters is adequate to detect more than 90 % of failures in many software systems, while 6-wise coverage is the maximum that is needed for practical purposes [17].

In this paper, we use the tool Testona (previously known as CTE-XL) [18] to generate CBS configurations. Testona implements the Classification Tree Method, a strategy for CIT. The method consists of tree steps: i) identify system's relevant aspects, and their corresponding values; ii) model a classification tree, where aspects are branches (*classifications*), and values are the leaves (*classes*); iii) combine classes from classifications into test cases (or configurations), covering different T-wise combinations.

III. Methodology

In this section, we present our methodology to generate test sequences. Our methodology consists of the three steps shown in Figure 3. 1) First, we model the aspects governing the elasticity behavior (*elasticity parameters*), such as the elasticity states, the thresholds for resource scale-out and scale-in, and the workload, into a Classification Tree (Section III-A). 2) Then, we generate a set of test configurations covering all the valid T-wise interactions between the elasticity parameters.

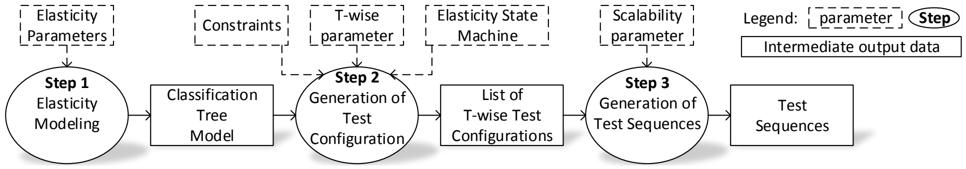
Elasticity Parameters
Constraints
T-wise parameter
Elasticity State Machine
Scalability parameter
Legend:
parameter
Step
Intermediate output data
Step 1 Elasticity Modeling
Classification Tree Model
Step 2 Generation of Test Configuration
List of T-wise Test Configurations
Step 3 Generation of Test Sequences
Test Sequences

Figure 3: Three Steps of Our Methodology

3) Finally, we generate test sequences covering all the possible transitions between the test configurations (Section III-C). Then, test sequences can be execute on a target CBS.

A. Elasticity Modeling

We use the Classification Tree Model (CTM) shown in Figure 4 (on the next page) to model the elasticity parameters that can be controlled during the test. Note that, for the purpose of this paper, we only consider a limited number of cloud infrastructure and benchmarking tool parameters. We leave the consideration of a more expressive model with additional elasticity parameters as future work.

The root of the CTM is the elasticity property, i. e., the CBS characteristic we want to investigate. We decompose elasticity into two main compositions, namely *cloud infrastructure*, which encompasses the parameters pertaining the CBS deployment, and *benchmark*, which models the workload of the system.

The cloud infrastructure is decomposed into the *elasticity_state* classification, and the *threshold* sub-composition containing classifications that represent scale-out (*scale_out_cpu_t*) and scale-in (*scale_in_cpu_t*) thresholds. The elasticity state classes represent the states of Figure 2. The classes in the classifications *scale_out_cpu_t* and *scale_in_cpu_t* refer to usage percentages of the currently allocated resources. For example, a 60 % scale-out threshold entails that the system switches to the scaling-out state when the CPU usage exceeds 60 %.

The *benchmark* composition is decomposed into the *workload_type* and *workload_intensity* classifications. In particular, the *workload_type* classes represent the three basic workload profiles consisting of read, write, or read and write operations. The *workload_intensity* classes represent two ways to drive the CBS through a scaling-out state: using a workload intensity that attempts to exhaust the allocated resource (*overloading*), or any other workload intensity (*non-overloading*).

B. Generation of Test Configurations

A *test configuration* is a set C of classes, where each atomic value of a classification, i. e., a leaf in the classification tree, is realized by a class [23]. For example, a test configuration *conf_i* can refer to the first class of each classification:

$$conf\_i = \{ready,\ 60\%,\ 10\%,\ read,\ overloading\}$$

Based on the CTM (Figure 4), we can create 162 ($3^4 \cdot 2^1$) configurations, where four classifications contain three classes, and one contains two classes.

However, each test configuration should also satisfy additional *cross-tree* constraints, which model particular aspects of the domain of testing CBS. For instance, we specify that a configuration in the *ready* or *scaling-in* state cannot have an *overloading* workload intensity. An overloading workload could unexpectedly trigger a resource scale-out while one is assessing CBS performance in those two states. Considering this constraint, the *conf_i* is an *invalid* configuration since it combines both, the *ready* and *overloading* classes.

T-wise combination: We propose the use of Combinatorial Interaction Testing [20] to test only T-wise combinations [14], [23] of elasticity parameters. This reduces the number of test configurations while ensuring variety in the CTM classes, in such a way that the number of configurations and their variety increase with the value of T. Considering the CTM of Figure 4, the value of T could range from 2 to 5 (the number of classifications in the CTM). While our methodology is independent of the value of T, we argue that considering pairwise interactions ($T = 2$) are sufficient for thoroughly testing CBS for performance degradations.

Table I lists all the configurations generated satisfying pairwise interactions of elasticity parameters and the constraints.

	elasticity _state	**scale _out _cpu _t**	**scale _in _cpu _t**	**workload _type**	**workload _intensity**
2w-conf_0	*scaling_in*	*90%*	*40%*	*read_write*	*non_overloading*
2w-conf_1	*scaling_out*	*90%*	*25%*	*write*	*overloading*
2w-conf_2	*scaling_out*	*75%*	*10%*	*read*	*non_overloading*
2w-conf_3	*ready*	*60%*	*25%*	*write*	*non_overloading*
2w-conf_4	*scaling_out*	*60%*	*40%*	*read*	*overloading*
2w-conf_5	*scaling_out*	*60%*	*10%*	*read_write*	*overloading*
2w-conf_6	*scaling_in*	*75%*	*25%*	*read_write*	*non_overloading*
2w-conf_7	*scaling_in*	*60%*	*10%*	*write*	*non_overloading*
2w-conf_8	*ready*	*90%*	*10%*	*read_write*	*non_overloading*
2w-conf_9	*ready*	*75%*	*40%*	*read*	*non_overloading*
2w-conf_10	*scaling_in*	*90%*	*25%*	*read*	*non_overloading*
2w-conf_11	*scaling_out*	*75%*	*40%*	*write*	*overloading*

Table I: The Twelve Pairwise Test Configurations

Note that only considering pairwise interactions, the number of test configurations is reduced from 162 (to cover all-wise interactions of elasticity parameters) to 12.

C. Generation of Test Sequences

In general, a test sequence is an ordered list of configurations covering all the possible reconfigurations, i. e., transitions between configurations. Creating an optimal sequence that covers all the reconfigurations without repeating them would require the use of a backtrack algorithm, facing an NP-complete problem.We choose to create several sequences, each one covering a subset of unique reconfigurations, and then we select several sequences covering together all the reconfigurations. The generation of test sequences is divided into three sub-steps: generation of a list of reconfigurations,

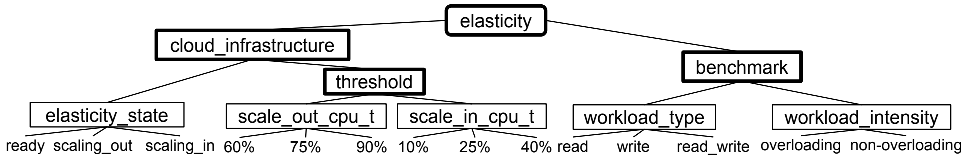

Figure 4: Classification Tree Model (CTM) of Elasticity Parameters

generation of a reconfiguration tree, and selection of a set of test sequences.

Generation of a list of reconfigurations: The reconfigurations should model the transitions in the *elasticity state machine* (Figure 1), where scaling-out and scaling-in states are always preceded or followed by a ready state. Among others, this allows CBS to stabilize itself after a resource change. Therefore, the sequence should only contain reconfigurations either to or from a ready state.

There are 54 reconfigurations between pairwise test configurations, which are partially shown in Table II. The table reports in the last column the change in the amount of resources allocated/deallocated, with respect to the following configuration in the sequence. For instance, $2w\text{-}reconf\_0$ and $2w\text{-}reconf\_3$ are reconfigurations towards a ready state ($2w\text{-}conf\_3$), when the number of resources is not changed ($= 0$). On the other hand, $2w\text{-}reconf\_9$ is a reconfiguration towards a scaling-in state ($2w\text{-}conf\_0$), when a resource is removed (-1), while during reconfigurations towards a scaling-out a resource is added ($+1$).

	previous configuration	next configuration	changes in the amount of resource
2w-reconf_0	2w-conf_0	2w-conf_3	0
...	...	...	...
2w-reconf_3	2w-conf_1	2w-conf_3	0
...	...	...	...
2w-reconf_9	2w-conf_3	2w-conf_0	-1
2w-reconf_10	2w-conf_3	2w-conf_1	+1
2w-reconf_11	2w-conf_3	2w-conf_2	+1
...	...	...	...

Table II: Excerpt of the 54 reconfigurations between pairwise test configurations.

Generation of a reconfiguration tree: Starting from the set of reconfigurations, it is possible to generate test sequences of any length by chaining reconfigurations. Figure 5 illustrates this concept using an example graph that considers an excerpt of the reconfigurations listed in Table II.

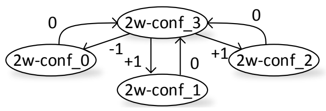

Figure 5: Excerpt of the Pairwise Reconfigurations Graph

The nodes of the graph represent test configurations, while the edges represent reconfigurations. Each edge is annotated with a number of resources allocated or deallocated during the reconfiguration, which corresponds to the last column of Table II. In this way, a test sequence can be seen as a path over this reconfigurations graph.

Reconfigurations are associated to changes in resource allocation, and hence, particular paths over the graph could lead to continuous de-allocation of resources, for example in the case of repeated scale-ins. Other paths could allocate too many resources, for example in the case of repeated scale-outs. In this paper, we consider assessing the performance with respect to elasticity, rather than scalability. Therefore, we bound the amount of resources (a) that can be allocated and deallocated using the three scalability parameters in Figure 2: the initial number of resource (i), the minimum number of resources (Min), and the maximum number of resources (Max).

To reduce the length of test sequences, we avoid to use the same reconfiguration several times by transforming the reconfiguration graph into a reconfiguration tree. The root can be any configuration associated to ready state. The other nodes in the tree are configurations reached through a sequence of unique reconfigurations (the edges), keeping resource allocation within the bound specified by *Min* and *Max*. For instance, in our experiments we consider $i = 1$ and $1 \leq a \leq 2$. With these parameters, it is not possible to move from the root (a ready state) to a scale-in configuration, because this would result in an amount of resources less than 1.

Figure 6 illustrates an example of reconfiguration tree from the graph of Figure 5. The root node corresponds to a configuration in the ready state, e. g., $2w\text{-}conf\_3$. On the first level (diamonds 1 and 2), $2w\text{-}conf\_0$ is not possible since it would lead to an allocated amount of resource lower than the minimum allowed. Then, only $2w\text{-}conf\_1$ and $2w\text{-}conf\_2$ can occur. On the third level of the left branch (diamond 3), $2w\text{-}conf\_1$ and $2w\text{-}conf\_2$ are not possible, since they would push the number of allocated resource over the maximum. Finally, on the sixth level (diamonds 4 and 5), the two sequences are terminated since all the reconfigurations (the six edges of each branch) were used.

Selection of a set of test sequences: Each branch of the reconfiguration tree (Figure 6) is a possible test sequence. However, a branch may not cover all the reconfigurations in the graph. Considering the longest branches we maximize the chances to cover all the reconfigurations. Therefore, as test

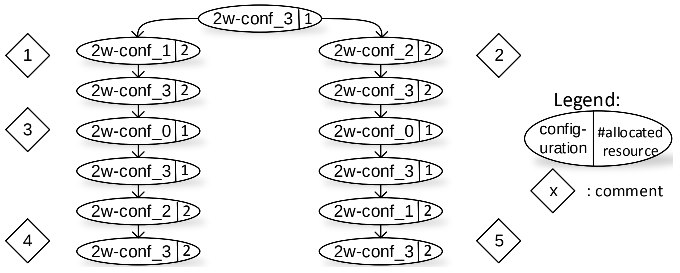

Figure 6: Reconfiguration Tree from the Excerpt of Pairwise Reconfigurations Graph

sequences we use the longest branches that together cover all the reconfigurations.

IV. Experiments

In this section, we present the experiment we conduct in order to investigate whether test sequences generated by our approach reveal performance degradations. In the experiment we generate test sequences with pairwise coverage of parameters. Indeed, if the smallest coverage reveals performance degradation, greater ones should also do so, as N-wise coverage includes M-wise coverage for $N > M$.

A. Experimental Setup

We use the document database MongoDB [2] as the case study for an elastic CBS, which is deployed as a sharding cluster [1], i.e., as a configuration server, a *mongos* instance, and several *shard* instances. The configuration server stores meta-data, while the *mongos* instance works as a coordinator and a load balancer routing queries and write operations to shards. Finally, the *shard* instances store and process the data in a distributed manner.

We execute all the experiments on Amazon Elastic Cloud Computing (EC2)[1]. We deploy mongos and configuration server instances on the same virtual machine ($m3.large$ type[2]), and each shard instance on dedicated virtual machines ($t2.small$ type). The initial MongoDB configuration consists of only one shard instance, and additional ones are allocated/deallocated during execution.

In another virtual machine ($c3.large$ type), we deploy the software artifacts for workload generation, i. e., a benchmark tool, and a workload controller. As benchmark tool, we use the Yahoo Cloud Serving Benchmark (YCSB) [5], which we use to generate initial data and the workload during the experiment. The workload controller is the software artifact that drives the workload generation based on test configuration's parameters by interacting to the benchmark tool.

B. Execution of Test Sequences

Each test sequence is independently executed, and for each one we (re-)deploy the CBS, avoiding any trash from previous executions. For each test sequence, we execute test configurations sequentially, respecting their parameters. We first set up the threshold on the cloud according to *threshold* value in the test configuration, then we generate the workload. The transition between configurations is determined by the ending of the *elasticity_state* associated to the test configuration (see Section II-A).

To generate the workload, we control the benchmark tool and parameterize it with the workload type and intensity specified by the test configuration parameters *workload_type*, *workload_intensity*, *thresholds* (*scale_out_cpu_t* and *scale_in_cpu_t*), and *elasticity_state*. The workload is steady throughout each test configuration, and varies only when the configuration ends.

[1]https://aws.amazon.com/ec2/

[2]https://aws.amazon.com/ec2/instance-types/

When the *workload_type* is read, we set up the benchmark tool to generate 100 % of read operations. For the *workload_type* read_write, we set up 50 % of read operations and 50 % of update operations. Finally, for the *workload_type* write, we set up 100 % of update operations. YCSB allows us to do it by using the native workload profiles[3] *workloadc* and *workloada* for read and read_write workload types. For the workload type write we customize a YCSB workload profile.

The last parameter we set up on the benchmark tool is the workload intensity, which is calculated after our previous work [4]. In particular, (1) we profile the resource usage of a workload type, and (2) we estimate the amount of work (workload intensity) necessary to either breach or stay below the scaling thresholds, depending on the elasticity state.

The second step of the workload intensity calculation involves the parameters *workload_intensity*, *thresholds*, and *elasticity_state* parameters. Due to the cross-tree constraints explained in Section III-B, the *elasticity_state* ready and scaling-in can only be associated to a non-overloading *workload_intensity*. When the *elasticity_state* is scaling_in, we set up a workload intensity that breaches the scale-in threshold. When the *elasticity_state* is ready, we set up a workload intensity that keeps the resource usage just below the scale-out threshold. For the *elasticity_state* scaling-out, we can have both types of *workload_intensity*, i. e., overloading and non-overloading. When the *workload_intensity* is non-overloading, we set up a value that just breaches the scale-out threshold. When the intensity is overloading, we set up a value that should use 100 % of CPU.

C. Test Oracle

In this experiment, we use a customized test oracle that assigns performance testing verdicts by using post-execution scripts. In the following, we explain the whole process of building and using the oracle, from the performance measurement to the assignment of test verdicts.

We measure CBS performance in number of operations answered per second, i. e., *throughput*, by calculating the average number of answers of each configuration in a test sequence. As explained in the previous section, for each configuration we generate a steady workload defined by an intensity (in operations per second). We expect that the average throughput measured for each configuration is close or equal to the generated workload.

The CBS throughput may be affected due to parameters we do not control, such as bandwidth or concurrent processes. Therefore, to ensure a more reliable experimental validation, we execute the same test sequence for 30 runs. Then, for each configuration c_i, we calculate the median throughput t_i per configuration over the 30 runs, and calculate its *percentage deviation* d_i compared to the workload w_i, defined as $d_i = \frac{t_i - w_i}{w_i}$. We refer to d_i as the *performance deviation* for the configuration c_i, which helps us to determine how close is the CBS performance compared to the expected result, i. e.,

[3]https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads

the workload. Note that, if a configuration is repeated in a test sequence, we consider a distinct performance deviation for each of its occurrences.

After calculating the performance deviations for each configuration, we use our test oracle to assign performance testing verdicts. For each configuration c_i in a test sequence, the *test oracle* compares the *expected performance* (i. e., workload w_i), to the absolute value of the *performance deviation* ($|d_i|$). Also note that, since the throughput is either less than or equal to the workload, the performance deviation is either 0 or negative, and hence we consider its absolute value. We define several *tolerance levels* (L) in the test oracle, where the higher is the tolerance, the higher is the performance deviation. This allows us, for instance, to investigate performance degradation only in reconfigurations whose throughput significantly deviates from the workload.

Finally, the test oracle assigns for each configuration c_i the *verdict* v_i as following:

$$v_i \begin{cases} \textit{pass} & \textit{if } |d_i| \leq L \\ \textit{fail} & \textit{if } |d_i| > L \end{cases}$$

If the absolute value of the performance deviation is less than or equal to the tolerance level, then the verdict is *pass*. Otherwise, the verdict is *fail*.

D. Experimental Results

We execute test sequences that cover all pairwise interactions between elasticity parameters. Our approach generates two test sequences: sequence 1 (*2w-TS1*) contains 50 reconfigurations, while sequence 2 (*2w-TS2*) contains 47 reconfigurations. Together, these test sequences contain 97 reconfigurations, covering all the allowed reconfigurations between pairwise configurations. Executing these 97 reconfigurations takes ≈ 6 h on the Amazon EC2 cluster, while repeating them for 30 times takes ≈ 180 h, i. e., ≈ 7.5 d.

We use our test oracle to test the performance of configurations, considering different tolerance levels that result in fail verdicts in a range of 100 % (0.05) to 0 % (0.35). In the industry, testers must decide which is the tolerance level that affects consumers. Figure 7 depicts the percentage of *fail* verdicts according to the tolerance level. We note that 100% of the verdicts are fail when the tolerance is at lowest level (no tolerance). Therefore, no configuration c_i achieves the ideal performance ($d_i = 0$) in the experiment, which is comprehensible as we are testing a distributed system under a massive sequence of reconfigurations. We also note that the percentage of fail verdicts decreases as the tolerance level increases, and that at tolerance level 0.35 there is no fail verdict. The most severe performance degradation happen at tolerance level 0.3.

1) Observations: Table III ranks configurations failing the test (i. e., *unstable* configurations) by the number of fail verdicts grouped by tolerance level, where configurations failing at the lowest level (0%) are omitted. Looking at these configurations in Table I, we see that all the unstable configurations

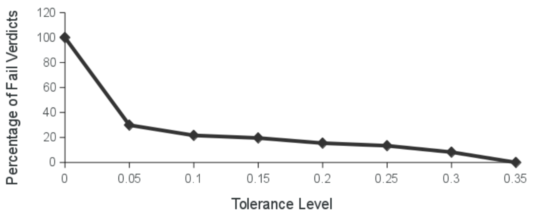

Figure 7: Percentage of Fail Verdicts by Tolerance

are related to the *ready* elasticity state. Furthermore, there is no pattern among the other parameters, despite for the *workload_type* that can only be non-overloading in ready states due to cross-tree constraints.

Tolerance	Fail Verdicts	unstable Configurations
0.30	8 (8%)	*2w-conf_8, 2w-conf_9*
0.25	13 (13%)	*2w-conf_8, 2w-conf_9*
0.20	15 (15%)	*2w-conf_8, 2w-conf_9*
0.15	19 (20%)	*2w-conf_8, 2w-conf_9*
0.10	21 (22%)	*2w-conf_8, 2w-conf_9, 2w-conf_3*
0.05	29 (30%)	*2w-conf_8, 2w-conf_9, 2w-conf_3*
0	97 (100%)	*

Table III: unstable Configurations Classified by Tolerance

We note that configurations from Table III tend to be unstable when they are preceded by specific configurations, and hence we highlight in Table IV the reconfigurations including unstable configurations. In the table, each level of tolerance includes the performance failures of the higher level, where tolerance 0.05 corresponds to all the reconfigurations of the table. Therefore, for each tolerance level, we only show new unstable reconfigurations not revealed by higher levels. Note that tolerance level 0.25 does not include any new unstable reconfiguration compared to level 0.30.

Tolerance	Reconfiguration	
	Previous Configuration	Next Configuration
0.30	*2w-conf_1*	*2w-conf_8, 2w-conf_9*
	2w-conf_2	*2w-conf_8, 2w-conf_9*
	2w-conf_4	*2w-conf_8, 2w-conf_9*
	2w-conf_5	*2w-conf_8*
	2w-conf_11	*2w-conf_8*
0.25	–	–
0.20	*2w-conf_5*	*2w-conf_9*
	2w-conf_11	*2w-conf_9*
0.15	*2w-conf_4*	*2w-conf_3*
	2w-conf_11	*2w-conf_3*
0.10	*2w-conf_5*	*2w-conf_3*
0.05	*2w-conf_1*	*2w-conf_3*
	2w-conf_2	*2w-conf_3*

Table IV: unstable Reconfigurations Classified by Tolerance

In Table IV, all the previous configurations are associated to scaling-out elasticity state, and all the configurations associated to a scaling-out elasticity state are listed in the table. Besides the elasticity state, there is no further parameter pattern among those configurations. As previously stated, the next configurations are all associated to the ready elasticity state. Crossing Table IV we see all the reconfigurations where the previous configuration is associated to scaling-out state and the next one to ready state. Therefore, we identify all the unstable reconfigurations.

Configuration *2w-conf_3* is only unstable when the tolerance level is higher than or equal to 0.2. On the other hand, configurations *2w-conf_8* and *2w-conf_9* are only unstable when the tolerance level is lower than or equal to 0.15. This is related to the parameter scaling-out threshold (*scale_out_cpu_t*), given it is higher for configurations *2w-conf_8* and *2w-conf_9* (90% and 75% of CPU) than for configuration *2w-conf_3* (60% of CPU). Then, CBS receives a workload with higher intensity, which makes the performance failure more severe.

Therefore, for the highest tolerance levels (0.30-0.20), we have the following reconfiguration pattern, where $*$ means any possible parameter value:

$$\begin{aligned} conf_{i-1} &= \{scaling\text{-}out,\ *,\ *,\ *,\ *\} \\ conf_i &= \{ready,\ \geq 75\%,\ *,\ *,\ *\} \end{aligned}$$

On the other hand, for the lowest tolerance levels (0.15-0.05), we have the following pattern:

$$\begin{aligned} conf_{i-1} &= \{scaling\text{-}out,\ *,\ *,\ *,\ *\} \\ conf_i &= \{ready,\ \geq 60\%,\ *,\ *,\ *\} \end{aligned}$$

Figure 8 depicts the throughput of each configuration in the test sequence *2w-TS1*, whereas *2w-TS2* presents the same behavior. The diamonds represent the workload, while box-and-whisker plots represent the distribution of each configuration throughput over 30 executions. At some configurations, the performance variation is very high, such as the configuration at the index 3. We see that such configurations have their medium values distant from the workload, what mean they are unstable. Those configurations match to the ones listed in the column *next configuration* of Table IV, when the tolerance level is high (0.30-0.20).

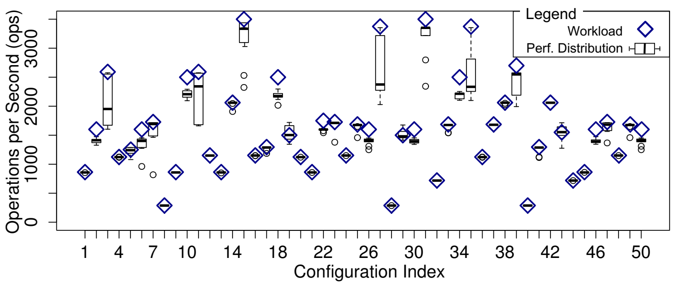

Figure 8: Throughput of Each Configuration in *2W-TS1*

2) Discussion: During the experiment execution we prevent resource exhaustion, and unbalanced data. We notice that performance degradation at unstable configurations occur due to load balancing problem. The newest shard, added during the scaling-out state that precedes the unstable configuration, does not receive as many requests as the existing one (oldest shard). Then, during a ready state that follows a scaling-out state, the oldest shard is exhausted.

Pairwise Adequacy: It could be argued that pairwise coverage may not be effective, and a greater coverage of elasticity parameters could find further performance degradations. However, executing sequences with such greater coverage is typically cost- and time-prohibitive. For instance, using three-wise coverage of elasticity parameters, only one combination greater than pairwise, results in 40 configurations. This sums up 2674 reconfigurations, which would take $\approx 7\,\mathrm{d}$ to be executed, ≈ 7 months when repeated for 30 times. Parallel execution could reduce the time needed to execute the sequences, but the whole process would still require a significant budget with a commercial cloud provider.

Preliminary experiments, where we only execute part of three-wise reconfigurations, suggest that test sequences achieving pairwise coverage reveal all the performance degradations of the case study. The three-wise test sequences we executed cover all the three-wise configurations, though they only cover a small part of their reconfigurations. Those test sequences do not reveal new unstable configurations or reconfigurations, where all of the unstable configurations found using three-wise follow the same pattern of pairwise. However, new assessments considering larger combination of elasticity parameters are left for future work.

V. Threats to Validity

In this paper, we restrain the list of elasticity parameters modeled into classification tree. We aim at most significant parameters, related to the elasticity behavior and workload generation. However, one could set up further parameters, such as a larger range of scale-out thresholds, what could help in getting more precise results of which is the exact threshold that results in more severe performance degradations. The parameters we use reveal significant performance degradations, what suggests that our approach works. Hence, for further analysis one could use the approach with a larger list of parameters.

MongoDB's shards are deployed on instances of type *t2.small*, which provide a limited amount of computing resource. For an industrial deployment, instances with more computing power could be more adequate. Indeed, depending on the workload seasonality, the elasticity changes could be too frequent when using instances with low computing resource. However, since we control the workload generation and only exhaust the CPU when it is required (overloading parameter), using *t2.small* instances is not a problem for the purpose of our experiment.

VI. Related Work

Gambi et al. [11] present foundational ideas for elastic computing systems by making parallels with elastic material. They suggest that techniques in testing elastic materials could be adapted to testing elastic computing systems. The authors go further in [10], and model elastic systems as a sequence of elasticity states (ES) called elastic transition sequences (ETS). Given an input workload they verify if the elasticity transition sequence executes as expected. However, the ETS do not consider scaling states (scale in/out) and ignore testing requirements such as coverage of workload or coverage of all possible sequences. The authors also present AUToCLES [9], a test-as-a-service (TaaS) tool. Given an input, the tool ensures the correct ETS is reached, however it is not very clear how they guarantee the reachability. Their resource management (elasticity control) allows customized reuse of virtual machines and resource sharing. In contrast, we consider elasticity as an infrastructure as a service (IaaS). Our elastic operations add/remove machines and respect the infrastructure's constraints such as time taken for virtual machine addition/removal.

There are several work on elasticity control which is peripheral to the subject of our work. Our goal is to test a cloud application covering various elasticity states and workload rather than develop autonomous control algorithms for elasticity of a CBS. We briefly mention work on elasticity control to give the reader an overview of a related area. Copil et al. [6] discuss Sybl a language to control elasticity, Han et al. [12] present a lightweight approach to resource scaling, Malkowski et al. [19] use empirical models of workloads for controlling elasticity. Truong et al. [24] present a platform as a service for elasticity control, and Dupont et al. [8] do experimental analysis on autonomic elastic control strategies. Finally, Islam et al. [15] present metrics for measuring elasticity on a cloud platform. In contrast, our work is about measuring the performance of a cloud application under varying workload and elasticity states.

Load testing addresses the problem of performance scalability but the authors do not address elasticity where the workload cannot only increase but also decrease. We hence explore combinatorial interaction testing [20] as a way to cover both variation in workload and elasticity states to produce trustworthy estimates of SLA and in particular performance of cloud applications. Our work is based on previous work on modeling [18] and generating test cases [22], [21] for CIT. In recent work [23], we go one step further and generate sequences of reconfigurations to evaluate reconfiguration impact in self-adaptive software systems. In this paper, we propose a graph-based algorithm to select reconfigurations.

VII. Conclusion

In this paper, we present an approach that creates short test sequences in order to identify configurations and reconfigurations that degrade CBS performance. We model elasticity parameters in a classification tree, generate a set of configurations with T-wise coverage of such parameters, and then generate test sequences covering the possible reconfigurations between configurations. Then, the test sequences are run, and CBS performance is measured.

In the experiment of this paper, we execute pairwise test sequences on MongoDB NoSQL database, a CBS case study. These test sequences reveal several significant performance degradations, and also allow us to identify a pattern for unstable reconfigurations. Preliminary comparison to results of partial three-wise test sequences, suggests that pairwise identifies all the performance degradations of our case study.

Our work is the first step on generating test sequences for performance testing of CBS. The preliminary results encourage future investigations. As future work, we plan to conduct a deeper study of which is the adequate coverage of elasticity parameters, and intend to analyze different performance parameters and CBS study cases.

References

[1] Amazon EC2 AutoScaling CoolDown, 2016. Available online at http://docs.aws.amazon.com/autoscaling/latest/userguide/Cooldown.html.

[2] MongoDB Web page, 2016. Available online at https://www.mongodb.com/.

[3] TESTONA Web page, 2016. Available online at http://www.testona.net/.

[4] Michel Albonico, Jean-Marie Mottu, and Gerson Sunyé. Controlling the elasticity of web applications on cloud computing. In *Proceedings of the 31st SAC*, 2016.

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM SoCC*, 2010.

[6] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Sybl: An extensible language for controlling elasticity in cloud applications. In *Proceedings of CCGrid*, 2013.

[7] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment*, 2011.

[8] Simon Dupont, Jonathan Lejeune, Frederico Alvares, and Thomas Ledoux. Experimental Analysis on Autonomic Strategies for Cloud Elasticity. 2015.

[9] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Automated testing of cloud-based elastic systems with AUToCLES. In *Proceedings of ASE 2013*, 2013.

[10] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Testing elastic systems with surrogate models. In *Workshop CMSBSE'13*, 2013.

[11] Alessio Gambi, Waldemar Hummer, Hong-Linh Truong, and Schahram Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 2013.

[12] Rui Han, Li Guo, Moustafa M Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *CCGrid 2012*, 2012.

[13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. *ICAC*, 2013.

[14] A. Hervieu, B. Baudry, and A. Gotlieb. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *22nd ISSRE*, 2011.

[15] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of ICPE'12*, 2012.

[16] Peter M Kruse and Joachim Wegener. Test sequence generation from classification trees. In *Proceedings of fifth ICST*, 2012.

[17] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo Jr. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, 2004.

[18] Eckard Lehmann and Joachim Wegener. Test Case by Means of the CTE XL. EuroSTAR 2000, 2000.

[19] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of ICAC '11*, 2011.

[20] Changhai Nie and Hareton Leung. A Survey of Combinatorial Testing. *ACM Comput. Surv.*, 2011.

[21] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 2012.

[22] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the third ICST*, 2010.

[23] Sagar Sen, Stefano Di Alesio, Dusica Marijan, and Arnab Sarkar. Evaluating reconfiguration impact in self-adaptive systems–an approach based on combinatorial interaction testing. In *Proceedings of SEAA*, 2015.

[24] Hong Linh Truong, S. Dustdar, G. Copil, A. Gambi, W. Hummer, Duc Hung Le, and D. Moldovan. CoMoT - A Platform-as-a-Service for Elasticity in the Cloud. In *Proceedings of IC2E*, 2014.