Towards Short Test Sequences for Performance Assessment of Elastic Cloud-based Systems

Michel Albonico and Paulo Varela
Federal Technological University - Paraná, Francisco Beltrão, Brazil

Keywords: Cloud Computing, Elasticity, Combinatorial Testing, Small Test Sequences, Performance Testing.

Abstract: Elasticity is one of the main features of cloud-based systems (CBS), helping them to meet performance requirements under a varying workload. Given the great number of combinations among workload and elastic adaptation parameters, assessing their effect on CBS performance may be prohibitive. Existing systematic combinatorial testing approaches can help to reduce such combinations, though most of them only consider conventional software architectures. In the literature, we only find a single work on elastic CBS combinatorial testing, presented by some of the authors. However, the paper only presents experimental results on 2-wise elasticity parameter interactions and shallowly explores the performance issue causes. In this paper, we lead a further experiment by using our previous approach to generate performance test cases that cover three elasticity parameter interactions (i. e., 3-wise), one interaction longer than on the previous paper. Despite the significant increase in execution time and cost, new experimental results do not reveal any new critical performance issue by *3-wise*, which enforces the acceptance of *2-wise* elasticity parameter interactions.

1 INTRODUCTION

In cloud computing, elasticity is a system property to adapt itself in response to workload variation (Herbst et al., 2013). The main goal of elasticity is for the cloud-based system (CBS) to meet the current workload demand as closely as possible, which guarantees an acceptable performance experience to the user. Adaptations (i.e., dynamic changes) in the CBS structure and its operational parameters help it to achieve elasticity when leading with computational resource variation.

Inadequate adaptations can result in CBS performance degradation, and consequently an unsatisfactory service to the user. Therefore, testers must assess CBS performance by testing their elasticity. A typical strategy to do so consists in varying the workload in a way the CBS adapts itself and assessing whether such adaptations lead to significant degradation in performance. However, the number of interactions between possible workload variations and CBS adaptations can be very large, where testing all of them can be exhaustive and expensive. Therefore, we must reduce the number of interactions, while still being possible to find performance degradation.

Some of the authors previous work (Albonico et al., 2017a) presents an approach based on Combinatorial Interaction Testing (CIT) to reduce the number of elasticity parameter interactions. The approach allows generating test configurations by combining a different number of elasticity parameters up to all parameter combination (*N-wise*). However, in the previous paper, we do not go further than two elasticity parameter combination (*2-wise*). In this paper, we focus on the impact of another elasticity parameter combination for the performance assessment of CBS systems.

Aiming at discovering an effective parameter coverage for performance assessment of elastic CBSs, we conduct two systematic experiments on a sharding deployment of MongoDB document database[1], this paper CBS case study: 1) the CBS was exposed to test sequences that cover 2 interactions (*2-wise*) among elasticity parameters; 2) we increase the number of parameters interactions by one, generating test sequences with 3 parameter interaction (*3-wise*), and then exposed the CBS to them.

Experimental results enforce it is acceptable to generate test sequences by covering *2-wise* interactions among elasticity parameters. Compared with *3-wise*, *2-wise* experiment reveals most of the elasticity-related performance issues, including the major ones,

[1]MongoDB web site: https://www.mongodb.com/

Albonico, M. and Varela, P.
Towards Short Test Sequences for Performance Assessment of Elastic Cloud-based Systems.
DOI: 10.5220/0007758501650175
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 165-175
ISBN: 978-989-758-365-0

for our case study. Furthermore, both experiments reveal the same problematic re-configuration (transition between two test configurations) pattern.

The remainder of this paper is organized as follows. Section 2 gives us a short introduction of cloud computing, its states and combinatorial testing. Section 3 presents the combinatorial-based test generation approach used in this paper. Section 4 describes the experiments and discusses the results. Section 5 discusses the related work. Finally, Section 6 concludes and lists future perspectives.

2 BACKGROUND

In this section, we describe the main aspects of elastic CBS and combinatorial interaction testing.

2.1 CBS Elasticity

Figure 1 presents an example of CBS exposition to cloud computing infrastructure elasticity (Albonico et al., 2017b).

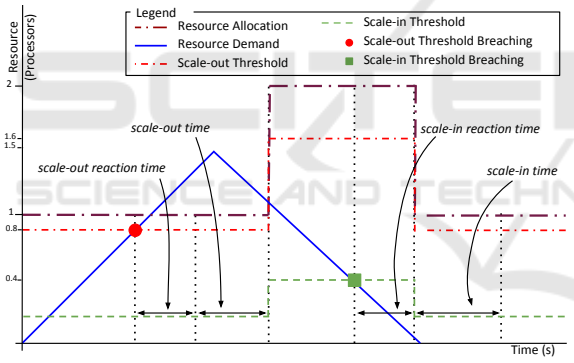


Figure 1: Typical Elastic Behavior.

The graph reports a *resource demand* (y-axis) expressed over time (x-axis) as a percentage of the currently allocated resources. Fancifully, the demand increases linearly, from 0 to 1.5, and then goes back to 0. Note that a resource demand equal to 1.5 means that the application demands 50 % more resources than the currently allocated ones.

If the resource demand exceeds the *scale-out threshold* (as a percentage of currently allocated resources, e.g. 80%) for the *scale-out reaction time*, the cloud elasticity controller instantiates a new resource, which becomes available after a *scale-out time* (the time for the cloud infrastructure 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* (as a percentage of currently allocated resources, e.g. 20%) for the *scale-in reaction time*, the cloud elasticity controller releases a resource. Note that, even if the infrastructure needs a *scale-in time* to actually release the resource, the resource is no longer available and the threshold values are updated as soon as the scale-in begins.

Cloud infrastructures can vary the number of computational resources according to demand, i. e., elasticity. In response to the cloud computing elasticity, the CBS adapts itself, transiting through three main states: *scaling-in*, *scaling-out*, and *ready*. Figure 2 depicts the transition between CBS elasticity states (Albonico et al., 2017b).

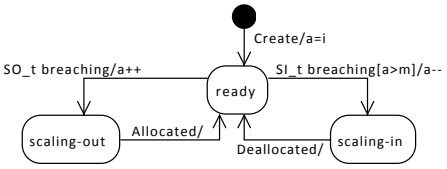


Figure 2: Elasticity State Transitions.

At the beginning, the CBS is launched (event *create*), and enters into the *ready* state, when the amount of allocated resource ($a = i$) is steady. When the resource demand breaches the scale-out threshold (*SO_t breaching* for a while (*scale-out reaction time*) and a new resource is already added (a++), the CBS enters into the *scaling-out* state, and remains there until reacting to the allocated resource. After a *scaling-out*, the CBS returns to the *ready* state. Then, when the resource demand breaches the scale-in threshold (*SI_t breaching*), and a resource is being deallocated (a–), the CBS enters into the *scaling-in* state. Note that the *scaling-out* state, the *scaling-in* state begins during the resource deallocation process since it is no longer. Finally, it returns to tge *ready* state.

2.2 Combinatorial 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 Test-

ing (CIT) is a strategy that consists of testing all T-wise combinations of the parameters of a system (Nie and Leung, 2011). 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 (Kuhn et al., 2004).

In this paper, we use the tool Testona (previously known as CTE-XL) (Lehmann and Wegener, 2000) to generate CBS configurations. Testona implements the Classification Tree Method, a strategy for CIT. The method consists of tree steps: i) identify system 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.

3 RE-USED TEST GENERATION APPROACH

In this section, we re-present the three-steps combinatorial-based approach to generate test sequences for elastic CBS (Albonico et al., 2017b) given its complexity.

3.1 Elasticity Modeling

We model the elasticity parameters that can be controlled during the test on a Classification Tree Model (CTM) (Figure 4). The root of the CTM is the elasticity, i. e., the CBS characteristic we want to investigate. We decompose it into two main compositions, namely *cloud_infrastructure*, which encompasses the parameters for the CBS deployment into the Cloud, and *benchmark*, which models the workload. These compositions are then decomposed into classifications. Additionally, the *cloud_infrastructure* has the sub-composition *threshold* that implements the scale-out (*scale_out_cpu_t*) and scale-in (*scale_in_cpu_t*), set on the Cloud.

The *cloud_infrastructure* composition is decomposed into the *elasticity_state* classification, which classes represent the possible CBS states (see Figure 2). The *scale_out_cpu_t* and *scale_in_cpu_t* classifications receive three different values (classes) each, where for the *scale_out_cpu_t* the value range is from 0 to 50, while for the *scale_in_cpu_t* it is from 50 to 100. Note that for avoiding bias, for each classification, we use values which are equally spaced. The *benchmark* composition is decomposed into the *workload_type* and *workload_intensity* classifications. In particular, *workload_type* classes represent the three basic workload profiles found in benchmark tools: read, write and read and write operations. The *workload_intensity* implements two workload intensities to drive the CBS through a scaling-out state: attempting to exhaust the allocated resource (*overloading*), and a fear workload, which never stresses the CBS (*non-overloading*)(Gambi et al., 2013b).

3.2 Test Configuration Generation

A *test configuration* is a set C of classes which are atomic values of classifications (leaves in the classification tree). For example, a test configuration *conf_i* is a configuration with the first class of each classification shown in the CTM of Figure 4:

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

Based on the CTM, we can create 162 ($= 3^4 \cdot 2^1$) configurations, where four classification has three classes, while one has two.

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 since this can unexpectedly trigger a resource scale-out. Considering this constraint, for instance, the *conf_i* is an *invalid* configuration since it combines both, the *ready* and *overloading* classes.

T-wise Combination

We use Combinatorial Interaction Testing (Nie and Leung, 2011) to test only T-wise combinations of elasticity parameters. This reduces the number of test configurations while ensuring variety in the CTM classes (Hervieu et al., 2011; Sen et al., 2015), where 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 this test generation methodology is independent of the value of T.

Table 1 lists the twelve *2-wise* test configurations, while Table 2 lists the forty *3-wise* configurations, where we can see a high variance. In these tables, each column represents a CTM classification, while the rows are their values (classes).

Table 1 lists all the configurations generated satisfying pairwise interactions of elasticity parameters

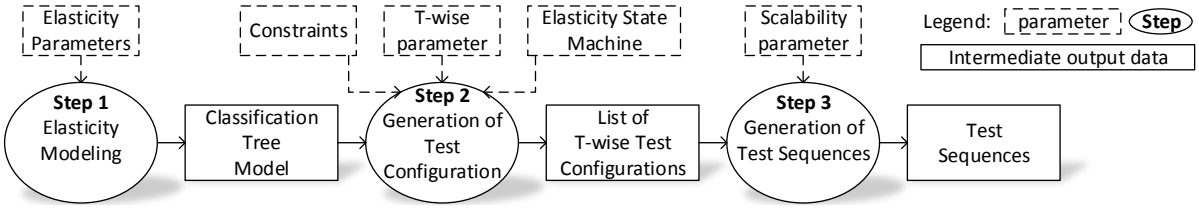


Figure 3: 3-Steps Approach (Albonico et al., 2017b).

Table 1: The 12 Pairwise Test Configurations (Albonico et al., 2017b).

	elasticity_state	scale_out_cpu_t	scale_in_cpu_t	workload_type	workload_intensity
2w-conf_0	*scaling_in*	*90%*	*40%*	*read_write*	*not_overloading*
2w-conf_1	*scaling_out*	*90%*	*25%*	*write*	*overloading*
2w-conf_2	*scaling_out*	*75%*	*10%*	*read*	*not_overloading*
2w-conf_3	*ready*	*60%*	*25%*	*write*	*not_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*	*not_overloading*
2w-conf_7	*scaling_in*	*60%*	*10%*	*write*	*not_overloading*
2w-conf_8	*ready*	*90%*	*10%*	*read_write*	*not_overloading*
2w-conf_9	*ready*	*75%*	*40%*	*read*	*not_overloading*
2w-conf_10	*scaling_in*	*90%*	*25%*	*read*	*not_overloading*
2w-conf_11	*scaling_out*	*75%*	*40%*	*write*	*overloading*

and the constraints. 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.

3.3 Test Sequence Generation

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

Generation of a List of Re-configurations

The re-configurations should model the elasticity state transitions (see Figure 2), 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.

There are 54 re-configurations between *2-wise* test configurations, which are partially shown in Table 3. Table last column reports the change in the amount of resource related to the next configuration. For instance, 2*w-reconf_0* and 2*w-reconf_3* are re-configurations towards a ready state (2*w-conf_3*), when the number of resources is not changed (=0), while 2*w-reconf_9* is a reconfiguration towards a scaling-in state (2*w-conf_0*), when a resource is removed (−1).

Table 2: The 40 3-wise Test Configurations.

	elasticity_state	scale_out_cpu_t	scale_in_cpu_t	workload_type	workload_intensity
3w-conf_0	scaling_in	90%	40%	read_write	not_overloading
3w-conf_1	scaling_in	75%	25%	write	not_overloading
3w-conf_2	scaling_in	60%	10%	read	not_overloading
3w-conf_3	scaling_out	90%	40%	write	overloading
3w-conf_4	scaling_out	90%	25%	read	not_overloading
3w-conf_5	scaling_out	75%	25%	read_write	overloading
3w-conf_6	ready	90%	10%	write	not_overloading
3w-conf_7	ready	75%	40%	read	not_overloading
3w-conf_8	ready	60%	25%	read_write	not_overloading
3w-conf_9	scaling_out	60%	10%	write	overloading
3w-conf_10	scaling_out	60%	40%	read	overloading
3w-conf_11	ready	60%	40%	write	not_overloading
3w-conf_12	scaling_in	75%	10%	read_write	not_overloading
3w-conf_13	scaling_out	60%	40%	read_write	not_overloading
3w-conf_14	scaling_out	75%	10%	read	not_overloading
3w-conf_15	scaling_out	90%	10%	read_write	overloading
3w-conf_16	scaling_out	60%	25%	write	not_overloading
3w-conf_17	scaling_out	75%	40%	write	overloading
3w-conf_18	scaling_in	60%	25%	read_write	not_overloading
3w-conf_19	ready	90%	25%	read	not_overloading
3w-conf_20	scaling_in	90%	10%	write	not_overloading
3w-conf_21	scaling_in	90%	25%	read	not_overloading
3w-conf_22	ready	75%	25%	write	not_overloading
3w-conf_23	ready	75%	10%	read_write	not_overloading
3w-conf_24	scaling_out	90%	25%	write	overloading
3w-conf_25	ready	90%	10%	read	not_overloading
3w-conf_26	ready	75%	40%	read_write	not_overloading
3w-conf_27	scaling_out	75%	10%	write	overloading
3w-conf_28	scaling_in	75%	25%	read	not_overloading
3w-conf_29	scaling_in	90%	40%	read	not_overloading
3w-conf_30	ready	60%	10%	read_write	not_overloading
3w-conf_31	ready	90%	25%	read_write	not_overloading
3w-conf_32	ready	60%	25%	read	not_overloading
3w-conf_33	scaling_out	60%	40%	read_write	overloading
3w-conf_34	scaling_out	75%	25%	read	overloading
3w-conf_35	scaling_out	90%	10%	read	overloading
3w-conf_36	scaling_in	60%	40%	write	not_overloading
3w-conf_37	ready	90%	40%	read_write	not_overloading
3w-conf_38	scaling_in	75%	40%	read_write	not_overloading
3w-conf_39	scaling_out	60%	25%	read_write	overloading

Table 3: Excerpt of the *2-wise* Re-configurations.

	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
...	...	...	...

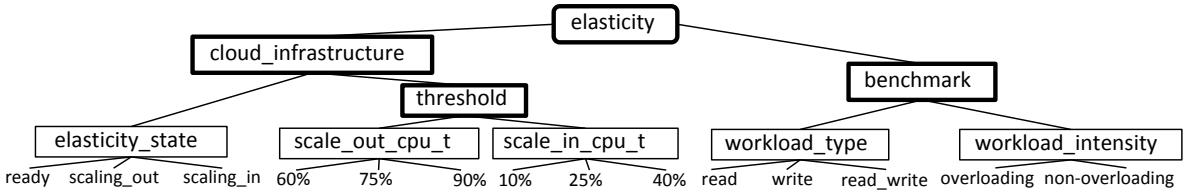


Figure 4: Elasticity Parameters CTM (Albonico et al., 2017b).

Generation of Test Sequences

From the set of re-configurations we can generate test sequences of any length by chaining re-configurations. Figure 5 illustrates this concept using an example graph that considers an excerpt of the re-configurations listed in Table 3.

The nodes of the graph are test configurations, while the edges are re-configurations. Each edge is annotated the value from the last column of Table 3. In this way, a test sequence can be seen as a path over this re-configurations graph.

Re-configurations are associated to changes in resource allocation, and hence, particular paths over the graph could lead to either continuous de-allocation of resources or allocate too many resources. In this paper, we bound the amount of resources (a) according to Figure 2, which avoids scalability bias: 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 re-configuration graph into a tree. The tree root can be any configuration associated to the ready state. The other nodes are configurations reached through a sequence of unique re-configurations, respecting resource amount bounds. In this paper, we consider the values $i = 1$ and $1 \leq a \leq 2$.

Figure 6 illustrates an example of a re-configuration tree from the graph of Figure 5. The test configuration 2*w-conf*_3, associated to the ready state, is the root node. Only 2*w-conf*_1 and 2*w-conf*_2 can occur at the first level (diamonds 1 and 2) since 2*w-conf*_0 would lead to an amount of resource lower than the minimum allowed. Test configurations 2*w-conf*_1 and 2*w-conf*_2 are not allowed at the third

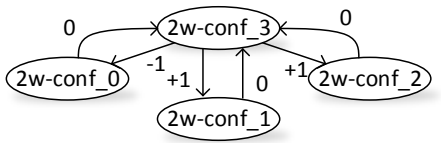


Figure 5: Excerpt of *2-wise* Re-configuration Graph.

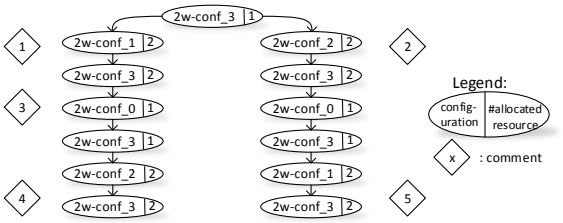


Figure 6: Reconfiguration Tree.

level of the left branch (diamond 3) since they would push the amount of resources over the maximum. Finally, at the lower level (diamonds 4 and 5), no new re-configuration is possible.

4 EXPERIMENTS

In this section, we present the experiment we conducted to investigate the necessary elasticity parameters coverage to find performance degradation into an elastic CBS. We generate two sets of test sequences, one by using *2-wise* coverage, and another by using *3-wise* coverage os elasticity parameters. Avoiding interference from not modeled parameters such as, bandwidth or concurrency, both sets of test sequences are re-executed 10 times. The number of repetitions respects financial constraints, once the experiments by using *3-wise* result in a high cost.

4.1 CBS Case Study

We use the document database MongoDB as a CBS, which is deployed as a sharding cluster [2]: a configuration server, a *mongos* instance, and several *shard* instances. Cluster sharding is useful when you need to distribute instances across several nodes. In the cluster, the configuration server stores meta-data, while the *mongos* instance works as a coordinator and a load balancer that routes queries and writes operations to shards. Finally, *shard* instances store and process the data in a distributed manner.

[2]MongoDB Website: https://www.mongodb.org/

4.2 Software Deployment

Figure 7 illustrates the deployment of both, MongoDB components and workload generation artifacts. We deploy all the software artifacts on Amazon EC2 all purpose virtual machines described in Table 4. We deploy the *mongos* and the *configuration server* on the same virtual machine (*t2.medium* type) (MongoS node), and the *shard* instance on a dedicated virtual machine (*t2.small* type) (MongoD node). The initial MongoDB configuration consists of only one shard instance, where additional instances are manually allocated/deallocated during execution by respecting the threshold classes in the test configurations. In another virtual machine (*t2.large* type) *(Workload Generator Node)*, we deploy the workload generation artifacts, i. e., the Yahoo Cloud Serving Benchmark (YCSB) (Cooper et al., 2010) as a benchmark tool, and the workload controller. During the experiment, the workload controller dynamically drives the benchmark tool according to the parameters in the test configurations.

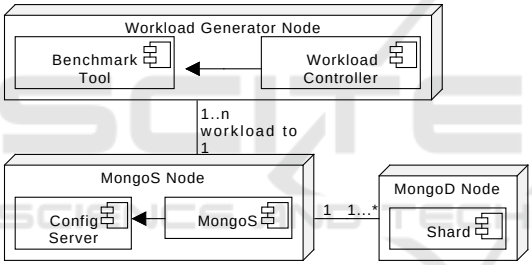


Figure 7: Software Components Deployment.

Table 4: Virtual Machine Configurations[3].

Machine Type	CPU	Memory	Disk	Price/Hour
t2.small	1 *vCPU (*3.3 GHz*)*	2 GB	10 GB	$0.0232
t2.medium	2 *vCPUs (*3.3 GHz*)*	4 GB	20 GB	$0.0464
t2.large	2 *vCPUs (*3.0 GHz*)*	8 GB	20 GB	$0.0928

4.3 Test Sequences Execution

Each test sequence is independently executed, and at each execution 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 with the test configuration (see Section 2.1).

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

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 type is the native profile of the benchmark tool. The workload intensity is steady throughout each test configuration and varies only when the configuration ends.

The workload intensity is calculated as our previous work (Albonico et al., 2016): (1) we profile the resource usage of a workload type, and (2) we estimate the number of request (workload intensity) to lead the CBS to the target elasticity state.

Due to the cross-tree constraints explained in Section 3.2, 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 a workload intensity that breaches the scale-in threshold. When the *elasticity_state* is ready, we set 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 a value that just breaches the scale-out threshold. When the intensity is overloading, we set a value that should use 100 % of CPU.

4.4 Performance Test Oracle

Ensuring a more reliable experimental validation, we repeat the same test sequence multiple times measuring the performance in the number of answered operations per second (i. e., *throughput*). Then, for each configuration c_i, we calculate its median throughput (t_i) over the executions, which is a parameter of the test oracle.

Figure 8 illustrates both, workload (dashed line) and configuration median throughput (solid line) variations, over the *3-wise* test sequence (*TS-0*) executions. In the figure, we see several throughput drops (negative different compared to the workload), such as the ones after index 10, which are easy to see. Nevertheless, other drops are less evident, such as the ones before the index 5. This illustrates different severity levels among throughput drops, which impacts on the consumers is difficult to estimate. Allowing testers to decide which is a critical level for their CBS, the performance test oracle enables a severity level setting.

The test oracle assigns performance testing verdicts by using post-execution scripts. For each configuration c_i, we calculate the *percentage deviation* (d_i)

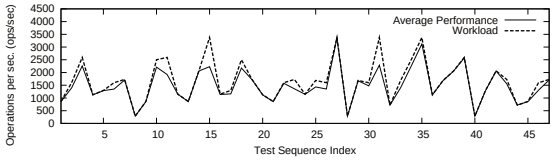


Figure 8: Performance Variation for *3-wise* Test Sequence*TS-0*.

compared to the workload w_i, defined as $d_i = \frac{t_i - w_i}{w_i}$. Note that, if a configuration is repeated in a test sequence, we consider a distinct performance deviation for each of its occurrences.

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* ($|d_i|$). Avoiding false positives due to light performance deviations, the test oracle supports several *tolerance levels* (L), where the higher is the tolerance, the higher is the performance deviation. Then, the test oracle assigns a *verdict* v_i to each configuration c_i as following:

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

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*.

4.5 Experimental Results

In this section, we first present the experiment results, and discuss them later, in Section 4.5.5.

4.5.1 Execution Time and Cost

Both measurements, execution time and cost, are important for testers to decide whether a test is prohibitive or not. Therefore, we present these measurements for *2-wise* and *3-wise* experiments.

Two-wise configurations result in 97 re-configurations. Executing them takes ≈6 h (≈3.8 s per configuration) on the Amazon EC2, while repeating them 10 times takes ≈60 h, costing *US*$ ≈20.88:

$$(2L + M + 5S) * T \quad (1)$$

where L, M and S are the costs per hour of dedicated *t2.large*, *t2.medium* and *t2.small* instances on Amazon EC2, and T is the execution time in hours. The L cost is multiplied the number of *t2.large* instances during the experiment, which is constant, while S is multiplied by the number of *t2.small* instances within one hour. Note that, on Amazon EC2, the shortest instance fee time is one hour. Therefore, the total of *t2.small* instances comes from the ≈ 3.7 minutes per configuration, resulting in ≈ 16 configurations per hour, where only 4 scale-out states in an additional instance. Nonetheless, two large and one medium instances live during the entire experiment, and therefore, only counts once in an hour.

The *3-wise* re-configurations are 2675, i. e., 27 × longer than for *2-wise*. Executing these once takes ≈170 h, and re-executing them 10 times takes ≈1900 h, i. e., ≈2.4 months. By executing them in parallel, one could reduce the execution time, however, it would still cost *US*$ ≈661.2.

4.5.2 Severity of Performance Issues

Aiming at investigating the statistical difference between *2-wise* and *3-wise* test sequences, we illustrate the distribution of their performance issue severity. As a performance issue severity, we consider the difference in operations per second between the workload and the measured performance, where higher is the difference, more severe is the performance issue.

Figure 9 illustrates the performance issue severity for *2-wise* test sequences. We see most of the performance issues with a severity value less than 600 *ops*, where the higher frequency is for severity values between 0 and 100 *ops*.

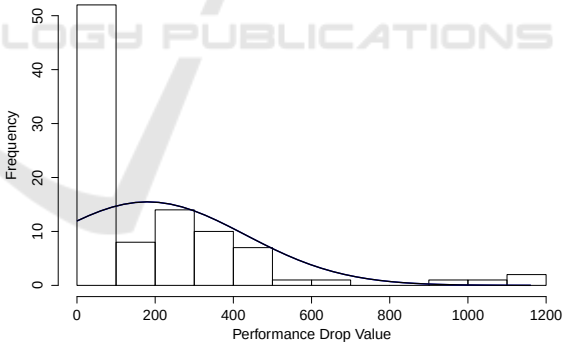


Figure 9: Distribution of the 2-wise Performance Issues.

Figure 10 illustrates the performance issue severity for *3-wise* test sequences. In the figure, most of the performance issues lie below 800*ops*, which is a little higher than in *2-wise* test sequences, and as well as for *2-wise* the higher frequency is also for severity values between 0 and 100. We also see that most of the performance issues are lower than 1500, except for some outliers that lie between 1500 and 2000. Note that the difference on the *frequency* axis labels of Figures 10 and 10 match the proportion in number of test re-configurations, i. e., 97 for *2-wise* and 2675 for *3-wise*.

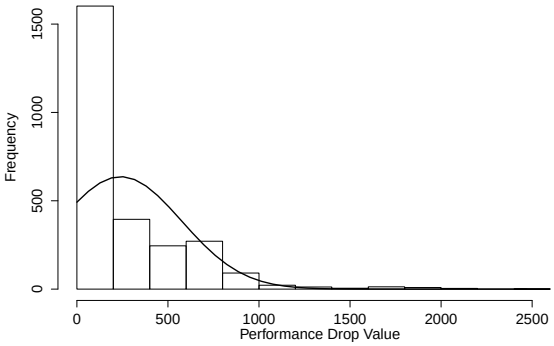


Figure 10: Distribution of the 3-wise Performance Issues.

4.5.3 Problematic Test Configurations

We use the test oracle (Section 4.4) to assess the performance of test configurations during the experiment. Considering different tolerance levels it results in fail verdicts in a range of 100 % (0.05) to 0 % (0.35). Despite the tolerance levels in this paper are realistic and could be used in industry, here we do not discuss which is the better level for performance testing of CBSs. We let it for testers to decide how they affect consumers.

In both experiments, with *2-wise* or *3-wise* coverage, we notice 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 since we are testing a distributed system under a massive sequence of re-configurations.

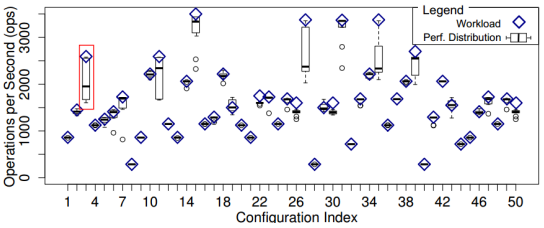
Table 5 groups by tolerance level *2-wise* and *3-wise* test configurations failing the test, which we call *unstable* configurations.

Table 5: Unstable Configurations.

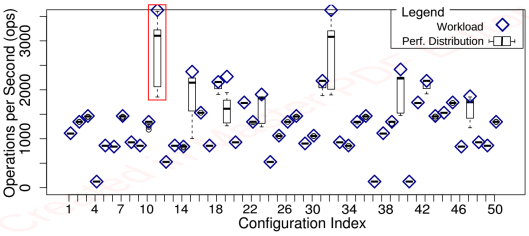
Tolerance	Unstable Configurations
0.30−0.15	*2w-conf_8, 2w-conf_9, 3w-conf_6, 3w-conf_7, 3w-conf_19, 3w-conf_22, 3w-conf_23, 3w-conf_25, 3w-conf_26, 3w-conf_31, 3w-conf_37*
0.10−0.05	*2w-conf_3, 3w-conf_8, 3w-conf_11, 2w-conf_30, 3w-conf_32*

Figure 11 illustrates such configurations.

In the figures, diamonds represent the workload, while box-and-whisker plots represent the distribution of each configuration throughput over 10 executions. At some configurations, such as at the index 3 in Figure 11a and at the index 11 in Figure 11b, the performance variation is very high. We see that such configurations have their medium values distant from the workload, what mean they are unstable. Those configurations match those listed in Table 5 with tolerance level between 0.30 and 0.15. Other configurations are less problematic, such as the ones at index 7 and at the index 23 in Figure 11b, which match to the test configurations in Table 5 with tolerance level between 0.10 and 0.05.



(a) *2-wise* Test Sequence *TS*-0



(b) *3-wise* Test Sequence *TS*-0

Figure 11: Throughput Variation.

4.5.4 Problematic Re-configurations

By traversing the test sequences executed in the experiments, we figured out the re-configurations that result in the *unstable* configurations listed in Table 5, which we call problematic re-configurations. For illustration reasons, Table 6 only depicts an excerpt of *2-wise* problematic re-configurations. Note that, for each tolerance level, we only show new unstable re-configurations not revealed by higher levels. Therefore, each level of tolerance includes the performance failures of the higher level, where tolerance 0.05 corresponds to all the re-configurations.

Table 6: Unstable Re-configurations by Tolerance.

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

In the table, all of the previous test configurations are associated with the scaling-out elasticity state, and all of the test configurations associated with a scaling-out elasticity state proceed at least a problematic configuration. As previously stated, the problematic con-

figurations are all associated with the ready elasticity state. This same pattern is found for both, *2-wise* and *3-wise* test sequences.

We also see that the severity of problematic configurations is linked to their workload intensity parameter, where the higher is the workload intensity, the more severe is the performance failure. For the highest tolerance level (0.30-0.20), we have test configurations associated with ready state and high workload intensity. For the lowest tolerance levels (0.15-0.05), we have test configurations associated with ready state and low workload intensity. Both are proceeded by a scale-out state.

4.5.5 Discussion

Before discussing the presented results, let us highlight that, during the experiment execution, we prevent resource exhaustion and unbalanced data between instances to avoid any interference to the elasticity parameters.

By comparing *2-wise* and *3-wise* experimental results, we see that grater parameter coverage is typically cost- and time-consuming. For instance, executing *3-wise* test sequences (only one combination grater than *2-wise*) once takes ≈170 h on the Amazon EC2, ≈28 times more than *2-wise*. One could argue that this can be solved by executing the test sequences in parallel. However, this does not reduce the execution cost, which for *3-wise* test sequences is also ≈28 more expensive.

Even though *3-wise* test sequences is more plural in therms of test configurations, we do not see any relevant difference in their performance issue severity distributions (Figures 9 and 10). This indicates the CBS behaves alike for both.

For the MongoDB case study, *2-wise* test sequences reveal the same pattern among problematic test re-configurations. Given the expensiveness of *3-wise* test sequences execution, we expected more severe performance degradation, which is not revealed in the experiment. We do not plan to execute test sequences with larger combination of elasticity parameters since the cost and execution time are prohibitive. However, one could re-execute the experiment by considering different or further elasticity parameters.

About the performance degradation found, we notice that they occur due to MongoDB load balancing problems. Right after being added during the scaling-out state that precedes unstable configurations,the newest shard node 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. That explains why only ready states with high intensity have severe performance degradation.

4.5.6 Threats to Validity

In this study, we consider that a reconfiguration can only occur by changing the elasticity states. One can argue that during an elasticity state the other parameters could be changed. It is right but useless in most situations. For instance, when the system is in a scaling_in state, it is useless to change the threshold or the workload_intensity parameters. Furthermore, as far as we experimented, we did not see major impact due to workload_type changes. In this paper, we only consider the elasticity performance issue, then the most interesting parameters is elasticity_state, whereas we are interested in measuring the impact of the other parameters when elasticity_state is changed.

We choose to have each reconfiguration only once per test sequence. Therefore, we need several sequences to cover all the re-configurations. An alternative would be to create an unique sequence with all the re-configurations. However, to satisfy all the constraints, we would need to duplicate the reconfigurations in the sequence. This solution could be employed in real life, but in this paper, to get executions that can be compared each other, we prefer to generate several sequences with unique reconfiguration in each one of them.

We restrict the list of elasticity parameters in our classification tree. More parameters result in test sequences longer than the ones we use in the experiments. We could set a wider range of scale-out thresholds. In particular, this would help us identifying which is the exact limitation of MongoDB at ready states. However, the paper focus on proposing an approach for elasticity testing of cloud systems, wherein the set parameters are enough to answer the research questions.

5 RELATED WORK

Gambi et al. (Gambi et al., 2013b) and model elastic systems as a sequence of elasticity states (ES) called elastic transition sequence (ETS). Given an input workload they verify if the elasticity transition sequence executes as expected. However, the ETS does not consider scaling states (scale in/out) and ignores testing requirements such as coverage of workload or coverage of all possible sequences. The authors also present AUToCLES (Gambi et al., 2013a), a test-as-a-service (TaaS) tool. Their resource management (elasticity control) allows customized reuse of virtual

machines and resource sharing, in contrast to usual Infrastructure-as-a-Service (IaaS). The elastic operations (add or remove virtual machines) we consider in this paper respect the infrastructure constraints. None of their work considers a strategy to soften the complexity of elasticity testing.

There are several research efforts 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 workloads 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. discuss Sybl a language to control elasticity (Copil et al., 2013), Han et al. present a lightweight approach for resource scaling (Han et al., 2012), and Malkowski et al. use empirical models of workloads for controlling elasticity (Malkowski et al., 2011). Albonico et al. (Albonico et al., 2016) present elasticity control of the specific case of web applications on the cloud. Finally, Truong et al. (Truong et al., 2014) present a platform as a service for elasticity control, and Dupont et al. do experimental analysis on autonomic elastic control strategies (Dupont et al., 2015). Finally, Islam et al. present metrics for measuring elasticity on a cloud platform (Islam et al., 2012). In contrast, our work measures the performance of a cloud application upon workload and elasticity states variations.

Systematic testing of a CBS for performance under elastic conditions (Brebner, 2012) is essential to guarantee service level agreements and be reliable under varying workloads. Our work is based on previous work on modeling (Lehmann and Wegener, 2000) and generating test cases (Perrouin et al., 2010; Perrouin et al., 2012) for CIT. In a recent work, Sen et al. (Sen et al., 2015) goes one step further and generates sequences of re-configurations to evaluate reconfiguration impact in self-adaptive software systems. In our previous work (Albonico et al., 2017b), we propose an approach to select re-configurations that represent realistic elasticity. However, we do not go further than test configurations that cover *2-wise* elasticity parameters.

6 CONCLUSION

In this paper, we parameterize a combinatorial-based approach to create *2-wise* and *3-wise* test sequences for elasticity testing. The approach is applied to assess the performance of a CBS case study, the MongoDB.

In the experiments, shortest test sequences, i. e., *2-wise*, reveal most of the performance degradation. It also allows us to identify a pattern for unstable re-configurations. Given the promising experimental results, and the large adoption of *2-wise* in standard software testing, we claim it is also an adequate coverage in the case of combinatorial test case generation for elastic CBS performance assessment. This is enforced by the presented high cost and long executions of *3-wise* or longer test sequences, which may make their executions impractical.

This work is our second step towards short test sequence generation for CBS performance assessment. The presented results enforce *2-wise* combinatorial testing as a combinatorial testing strategy. However, one cloud compare other methods, as well as further elasticity parameters, case studies, and scalability (more than two nodes). As future work, we plan to compare this paper CIT to further test case generation strategies. We also plan to conduct a deeper evaluation of elasticity parameters, scalability and case studies.

REFERENCES

Albonico, M., Alesio, S. D., Mottu, J., Sen, S., and Sunyé, G. (2017a). Generating Test Sequences to Assess the Performance of Elastic Cloud-Based Systems. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 383–390.

Albonico, M., Di Alesio, S., Mottu, J., Sen, S., and Sunyé, G. (2017b). Generating test sequences to assess the performance of elastic cloud-based systems. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, June 25-30, 2017*, pages 383–390.

Albonico, M., Mottu, J.-M., and Sunyé, G. (2016). Controlling the elasticity of web applications on cloud computing. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 816–819.

Brebner, P. C. (2012). Is your cloud elastic enough?: performance modelling the elasticity of infrastructure as a service (iaas) cloud applications. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 263–266. ACM.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SoCC'10*, New York, NY, USA. ACM.

Copil, G., Moldovan, D., Truong, H.-L., and Dustdar, S. (2013). Sybl: An extensible language for controlling elasticity in cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 112–119. IEEE.

Dupont, S., Lejeune, J., Alvares, F., and Ledoux, T. (2015). Experimental Analysis on Autonomic Strategies for Cloud Elasticity.

Gambi, A., Hummer, W., and Dustdar, S. (2013a). Automated testing of cloud-based elastic systems with AUToCLES. In *The proceedings of ASE'13*, pages 714–717. IEEE/ACM.

Gambi, A., Hummer, W., and Dustdar, S. (2013b). Testing elastic systems with surrogate models. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 8–11. IEEE.

Han, R., Guo, L., Ghanem, M. M., and Guo, Y. (2012). Lightweight resource scaling for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 644–651. IEEE.

Herbst, N. R., Kounev, S., and Reussner, R. (2013). Elasticity in Cloud Computing: What It Is, and What It Is Not. *ICAC*.

Hervieu, A., Baudry, B., and Gotlieb, A. (2011). PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 120–129.

Islam, S., Lee, K., Fekete, A., and Liu, A. (2012). How a consumer can measure elasticity for cloud platforms. In *Proceedings of ICPE'12*, page 85, New York, New York, USA. ACM Press.

Kuhn, D. R., Wallace, D. R., and Gallo Jr, A. M. (2004). Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, 30(6):418–421.

Lehmann, E. and Wegener, J. (2000). Test Case by Means of the CTE XL. EuroSTAR 2000.

Malkowski, S. J., Hedwig, M., Li, J., Pu, C., and Neumann, D. (2011). Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, page 131, New York, New York, USA. ACM Press.

Nie, C. and Leung, H. (2011). A Survey of Combinatorial Testing. *ACM Comput. Surv.*, 43(2).

Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., and Traon, Y. (2012). Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal*, 20(3-4):605–643.

Perrouin, G., Sen, S., Klein, J., Baudry, B., and Le Traon, Y. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 459–468. IEEE.

Sen, S., Alesio, S. D., Marijan, D., and Sarkar, A. (2015). Evaluating Reconfiguration Impact in Self-Adaptive Systems – An Approach Based on Combinatorial Interaction Testing. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 250–254.

Truong, H. L., Dustdar, S., Copil, G., Gambi, A., Hummer, W., Le, D. H., and Moldovan, D. (2014). CoMoT - A Platform-as-a-Service for Elasticity in the Cloud. In *Proceedings of IC2E*.