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 database1, 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,

---

1MongoDB web site: https://www.mongodb.com/
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.](image)

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. 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, transitioning 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.](image)

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 the 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 $2^T \cdot \left(\begin{array}{c} n \\end{array}\right)$ 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) and scale-in (scale_in_cpu), 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, and scale_in_cpu classifications receive three different values (classes) each, where for the scale_out_cpu the value range is from 0 to 50, while for the scale_in_cpu 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:

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

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

<table>
<thead>
<tr>
<th>elasticity</th>
<th>in</th>
<th>out</th>
<th>cpu</th>
<th>cpu</th>
<th>workload</th>
<th>workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>2w-conf1</td>
<td>90%</td>
<td>10%</td>
<td>read</td>
<td>write</td>
<td>not-overloading</td>
<td></td>
</tr>
<tr>
<td>2w-conf2</td>
<td>75%</td>
<td>25%</td>
<td>write</td>
<td>not-overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2w-conf3</td>
<td>60%</td>
<td>40%</td>
<td>write</td>
<td>not-overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2w-conf4</td>
<td>90%</td>
<td>10%</td>
<td>read</td>
<td>overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2w-conf5</td>
<td>75%</td>
<td>25%</td>
<td>read</td>
<td>write</td>
<td>overloading</td>
<td></td>
</tr>
<tr>
<td>2w-conf6</td>
<td>60%</td>
<td>40%</td>
<td>write</td>
<td>overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2w-conf7</td>
<td>90%</td>
<td>10%</td>
<td>read</td>
<td>not-overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2w-conf8</td>
<td>75%</td>
<td>25%</td>
<td>read</td>
<td>not-overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2w-conf9</td>
<td>60%</td>
<td>40%</td>
<td>read</td>
<td>write</td>
<td>overloading</td>
<td></td>
</tr>
<tr>
<td>2w-conf10</td>
<td>90%</td>
<td>10%</td>
<td>read</td>
<td>not-overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2w-conf11</td>
<td>75%</td>
<td>25%</td>
<td>read</td>
<td>write</td>
<td>overloading</td>
<td></td>
</tr>
</tbody>
</table>

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, 2w-reconf_0 and 2w-reconf_3 are re-configurations towards a ready state (2w-conf_3), when the number of resources is not changed (=0), while 2w-reconf_9 is a reconfiguration towards a scaling-in state (2w-conf_0), when a resource is removed (−1).
Towards Short Test Sequences for Performance Assessment of Elastic Cloud-based Systems

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 with 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 \((\text{Min})\), and the maximum number of resources \((\text{Max})\).

To reduce the length of test sequences, we avoid using 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 2w-conf_3, associated to the ready state, is the root node. Only 2w-conf_1 and 2w-conf_2 can occur at the first level (diamonds 1 and 2) since 2w-conf_0 would lead to an amount of resource lower than the minimum allowed. Test configurations 2w-conf_1 and 2w-conf_2 are not allowed at the third 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 of 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 \(^\text{2}\); 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.](image)

Table 4: Virtual Machine Configurations³.

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>CPU</th>
<th>Memory</th>
<th>Disk</th>
<th>Price/Hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.small</td>
<td>1 vCPU</td>
<td>2 GB</td>
<td>10 GB</td>
<td>$0.0232</td>
</tr>
<tr>
<td>t2.medium</td>
<td>2 vCPUs</td>
<td>4 GB</td>
<td>20 GB</td>
<td>$0.0464</td>
</tr>
<tr>
<td>t2.large</td>
<td>2 vCPUs</td>
<td>8 GB</td>
<td>20 GB</td>
<td>$0.0928</td>
</tr>
</tbody>
</table>

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).

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. 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 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 ci, we calculate its median throughput (τi) 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 ci, we calculate the percentage deviation (di)
Towards Short Test Sequences for Performance Assessment of Elastic Cloud-based Systems

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 ≈6h (≈3.8s per configuration) on the Amazon EC2, while repeating them 10 times takes ≈60h, costing US$≈20.88:

\[(2L + M + S) \times T \]  

(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 \( \approx 3.7 \) minutes per configuration, resulting in \( \approx 16 \) 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., \( 27 \times \) longer than for 2-wise. Executing these once takes \( \approx 170h \), and re-executing them 10 times takes \( \approx 1900h \), i.e., \( \approx 2.4 \) months. By executing them in parallel, one could reduce the execution time, however, it would still cost US$\approx661.2.

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 10 illustrates the performance issue severity for 3-wise test sequences. In the figure, most of the performance issues lie below 800ops, 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.
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>
<thead>
<tr>
<th>Tolerance</th>
<th>Unstable Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.30–0.15</td>
<td>2w-conf 8, 2w-conf 9, 3w-conf 6, 3w-conf 7, 3w-conf 9, 3w-conf 22, 3w-conf 23, 3w-conf 25, 3w-conf 26, 3w-conf 31, 3w-conf 37</td>
</tr>
<tr>
<td>0.10–0.05</td>
<td>2w-conf 3, 3w-conf 8, 3w-conf 11, 2w-conf 30, 3w-conf 33</td>
</tr>
</tbody>
</table>

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.

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>
<thead>
<tr>
<th>Reconfiguration</th>
<th>Previous Configurations</th>
<th>Problematic Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.30</td>
<td>2w-conf 3, 2w-conf 4, 3w-conf 11, 2w-conf 2, 2w-conf 11</td>
<td>2w-conf 3, 2w-conf 4, 3w-conf 11</td>
</tr>
<tr>
<td>0.25</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>0.20</td>
<td>2w-conf 5, 2w-conf 11, 2w-conf 2</td>
<td>2w-conf 5, 2w-conf 11</td>
</tr>
<tr>
<td>0.15</td>
<td>2w-conf 4, 2w-conf 11, 2w-conf 2</td>
<td>2w-conf 4, 2w-conf 11</td>
</tr>
<tr>
<td>0.10</td>
<td>2w-conf 5, 2w-conf 11, 2w-conf 2</td>
<td>2w-conf 5, 2w-conf 11</td>
</tr>
<tr>
<td>0.05</td>
<td>2w-conf 3, 2w-conf 2</td>
<td>2w-conf 3, 2w-conf 2</td>
</tr>
</tbody>
</table>

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 are more plural in therm 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 re-configurations 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).

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


