Finding Unsafe Heterogeneous Configuration in Cloud Systems

Anonymous Author(s)
Submission Id: 353

Abstract

There is an increasing demand for heterogeneous configurations due to heterogeneous hardware and due to the requirement of online re-configuration. However, allowing different nodes to have different configurations may cause errors when these nodes communicate, even if the configuration of each node contains valid values.

To test which configuration parameters are unsafe when configured in a heterogeneous manner, this work re-uses existing unit tests but run them with heterogeneous configurations. To address the challenge that unit tests often share configuration across different nodes, we incorporate a number of heuristics to accurately map configuration objects to nodes. To address the challenge that there are too many tests to run, we pre-run unit tests to determine effective unit tests for each configuration parameter and further introduce pooled testing to test several parameters together. Our evaluation finds 27 unsafe heterogeneous configuration parameters in HDFS, HBase, YARN, and MapReduce. We further propose solutions for a subset of them.

ACM Reference Format:
Anonymous Author(s). 2020. Finding Unsafe Heterogeneous Configuration in Cloud Systems. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

While many distributed systems were initially designed under the assumption that all nodes share the same configuration, heterogeneous configuration becomes increasingly popular for two reasons: first, heterogeneous hardware naturally calls for a heterogeneous configuration to achieve optimal performance [5, 20, 23, 36]; second, even for a homogeneous system, sometimes we need to change its configuration at runtime to adapt to the workload [5, 21, 25, 29], but rebooting the whole system with a new configuration is sometimes too disruptive. To solve this problem, a number of works propose to incrementally reboot a subset of nodes with the new configuration [25], until all nodes have the new configuration. Both these two cases may cause different nodes to have different configurations, either permanently or temporarily.

Heterogeneous configuration, however, may cause the system to fail if not used properly. For example, if one node is configured to encrypt its communication channel while the other node is not, then obviously the communication will fail. This type of errors is different from the typical configuration errors caused by [2, 28, 30, 31, 35, 37]: in our case, both configuration values (i.e., using or not using encryption) are valid; the problem is caused by two nodes with different configurations communicating with each other.

The goal of this paper is to investigate, in real-world applications, how many of their configuration parameters cannot be set in a heterogeneous manner. We call them *unsafe heterogeneous configuration parameters* in this paper.

At a high level, our investigation is not much different from classic program testing: we test the target application with different heterogeneous configurations and different inputs to see whether the application will fail. However, this method also meets the classic challenge: a particular configuration parameter may only take effect when a particular piece of code is executed, so to test the parameter, we need to drive the application to a perhaps corner case.

To address this challenge, we observe that mature applications usually have well-designed unit tests, which have already considered this problem: to test the effects of a certain configuration parameter, they generate inputs so that the particular configuration will take effect and have rules to check whether the applications are in healthy states. Following this observation, we utilize existing unit tests by assigning different configurations to different nodes in these tests.

We meet two technical challenges when applying this idea. First, to run a unit test with a heterogeneous configuration, we need to be able to assign different configuration values to different nodes. This is trivial in a real distributed setting since we can give different configuration files to different nodes. This task, however, is significantly more challenging in unit tests, which often create nodes as threads within a process: on the one hand, a unit test may create a configuration object and share it with different nodes; on the other hand, a node may have sub-components, which may create their own configuration objects. Both make it harder to map a configuration object to a particular node. To address this problem with minimal instrumentation effort, we incorporate a number of heuristics to identify configuration sharing—we will clone the configuration object in this case—and infer the mapping from configuration objects to nodes.

The second challenge is the large number of tests to run. To alleviate this problem, we incorporate several techniques: 1) we pre-run unit tests to identify which configuration parameters are used by what type of nodes in each unit test, so that we won't try to assign a parameter to a node that will

not use the parameter; 2) under the assumption that most parameters are safe, we introduce *pooled testing*, which tests several parameters together with one unit test, and separates them only if the pooled test fails.

Following these ideas, we have built ZebraConf, a framework to run unit tests under different heterogeneous configurations. We have applied ZebraConf to HDFS [9], YARN [32], MapReduce [27], and HBase [1].

- ZebraConf reports a total number of 58 unsafe parameters in these applications. Our manual analysis shows 27 of them are truly unsafe parameters, 4 of them are caused by errors unrelated to heterogeneous configuration but happened to be triggered by our tests, and the remaining ones are false positives. While many of these unsafe parameters are expected (e.g., parameters related to encryption, compression, heartbeat, etc), some of them are more subtle. For example, we find setting a heterogeneous bandwidth limitation on different DataNodes in HDFS may cause one DataNode with a high limit to overload a DataNode with a low limit, so that the latter cannot send heartbeats in time and eventually be identified as dead. We further propose solutions for a subset of these parameters.
- With its heuristics, ZebraConf's can correctly map configuration objects to different nodes in 89.3% to 98.4% of the unit tests. It only requires to instrument 21 to 36 lines of code in different applications.
- Pre-running tests and pooled testing have reduced the total number of units tests to run by two to three orders of magnitude. As a result, we are able to run all tests on a cluster of 100 machines within 41 hours. While these numbers are certainly not small, they are affordable since we do not need to run these tests frequently.

The rest of the paper proceeds as follows. Section 2 presents the related work, which motivates our work. Section 3 gives an overview of work. Section 4 to Section 6 each introduces the design of one key component of ZebraConf. Section 7 presents the results of our evaluation and Section 8 concludes the paper.

2 Related Work and Motivation

While many distributed systems were initially designed under the assumption that all nodes have the same configuration (i.e., homogeneous configuration), heterogeneous configuration becomes increasingly popular for several reasons.

Heterogenous hardware. Heterogenous hardware naturally calls for a heterogeneous configuration to achieve best performance [5, 20, 23, 36]. While many systems still require a static configuration when booting the system, a number of systems (e.g., HDFS, HBase, Reddis, Kafka) even allow the administrator to change certain configuration parameters at runtime. For example, developers made parameter 'dfs.datanode.balance.bandwidthPerSec' online reconfigurable from HDFS 0.20 [12], because administrators found its optimal value is almost never known at system booting time. Either setting bandwidthPerSec too low or too high may bring the cluster into a "maintenance window", which is expensive and a serious maintenance problem for large clusters.

Re-configuring a homogeneous system. Even for a homogeneous system, sometimes it's beneficial to change its configuration to adapt to the workload [5, 21, 25, 29]. While rebooting the whole system with a new configuration file is always possible [3, 7, 18, 22, 24, 38], it is often too disruptive especially for a large cluster. To solve this problem, recent works propose to incrementally reboot a subset of the nodes with the new configuration, till all nodes have the new configuration [25]. This approach will create a short window of heterogeneous configuration for the whole system.

However, heterogeneous configuration may lead to correctness issues when nodes with different configuration values communicate. Some of these issues are obvious: if a node is configured to encrypt its data and another node is not aware that data is encrypted, they cannot communicate properly. Some of these issues are more subtle and perhaps unexpected as shown in our evaluation. The goal of this work is to study how many configuration parameters in real applications are unsafe to be configured in a heterogeneous manner.

Finding configuration errors. There is a substantial amount of work targeting identifying configuration errors caused by invalid configuration values (including but not limited to [2, 19, 28, 30, 31, 35, 37]). For example, ConfValley [19] defines a systematic way to validate configuration values; PCheck [31] extracts application code that checks the validity of configuration values and executes such code before deploying a configuration.

Our work is different from these works because a parameter can be unsafe in a heterogeneous setting even if all its values at different nodes are valid (e.g. one node is configured to encrypt data and another node is configured not to encrypt data). Therefore, it is impossible to check such problems locally at one node.

3 Overview

3.1 Goal

This work targets a distributed system, which is composed of a number of nodes (i.e. processes) connected by the network. We assume each node can be configured independently with its own configuration file. The goal of this work is to investigate whether assigning different configuration files to different nodes will cause errors.

We assume each configuration file contains valid values, i.e., if all nodes use the same configuration file, the system will be in correct state. We assume, within one node, all code always sees the same configuration. In other words, issues caused by incorrect implementation of online re-configuration

(e.g., missing updates to some cached/affected variables) are out of the scope of our work.

We formally define our goal as follows:

1. F denotes a configuration file.
2. $HomoConf(F)$ denotes a homogeneous configuration, in which all nodes have the same configuration file F.
3. $HeterConf(F_1, ..., F_n)$ denotes a heterogeneous configuration, in which node i has configuration file F_i.
4. I denotes a sequence of inputs to the target application. Apart from explicit inputs through the application APIs, all nondeterministic factors, such as timing and randomness, as modeled as implicit inputs as well.
5. A testing *oracle* can verify whether the application is in a correct state given I and either a $HomoConf(F)$ or a $HeterConf(F_1, ..., F_n)$.

Definition 3.1 (Unsafe heterogeneous Configurations)**.** *We say $HeterConf(F_1, ..., F_n)$ is unsafe, if $\exists I$, such that, $oracle(I \cdot HeterConf(F_1, ..., F_n))$ is false, but $\forall$ i from 1 to n, $oracle(I \cdot HomoConf(F_i))$ is true.*

Intuitively, this means that the heterogeneous configuration will cause problems even if all the individual configuration files are valid.

3.2 Our Approach

To understand whether certain heterogeneous configurations are safe, we use the traditional software testing approach: we generate a number of heterogeneous configurations, run the target application with these heterogeneous configurations and different inputs, and check whether the application will run into errors. However, the challenge of this approach is that a particular configuration parameter may only take effect when a corner piece of code is executed and thus when testing the parameter, we need to generate specific inputs to drive the application to the corner case .

To address this challenge, we utilize existing unit tests built by the application developers: we run these unit tests with the corresponding heterogeneous configuration. Since the unit tests of a mature application should cover most of the code of the application, they naturally provide the ability to drive the application to different corner cases and test whether the application is in a correct state. In other words, we assume the unit tests can provide the input I and approximate the *oracle* in Definition 3.1. Following this idea, we have built ZebraConf, a framework to generate heterogeneous configurations, to run unit tests with these heterogeneous configurations, and to instrument the target application to facilitate such testing.

3.3 System Architecture

As illustrated in Figure 1, ZebraConf consists of three key components: ConfAgent, TestRunner and TestGenerator.

At the top layer, TestGenerator determines which unit tests to run and what heterogeneous configuration to use for each unit test.

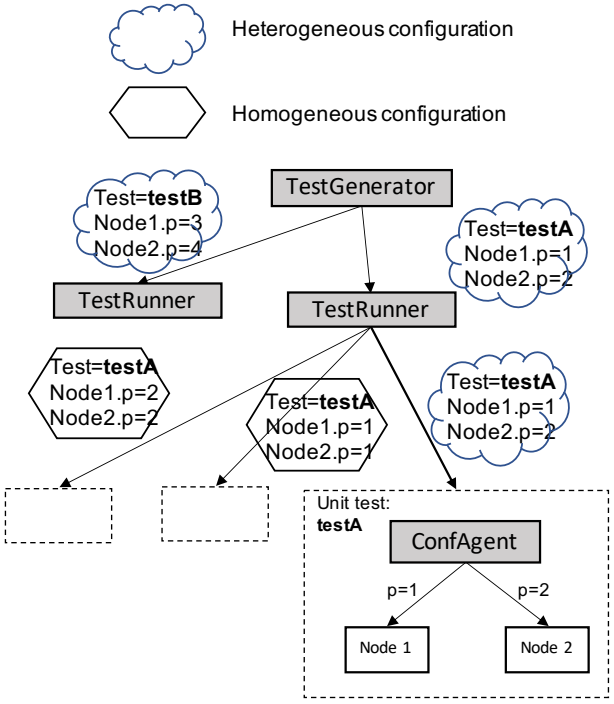

Figure 1. Overview of ZebraConf.

At the middle layer, given a unit test and a heterogeneous configuration, TestRunner follows Definition 3.1 to test 1) whether the unit test will report an error for the given heterogeneous configuration; and 2) whether the unit test will report an error for each corresponding homogeneous configuration.

At the bottom layer, ConfAgent is responsible for running a given unit test with a given configuration, either heterogeneous or homogeneous. The main task of ConfAgent is to distribute different configurations to different nodes.

4 Design of ConfAgent

In ZebraConf, ConfAgent is responsible for running a unit test with a given configuration. Since the major challenge comes from heterogeneous configuration, we will mainly discuss this case in this section with an example shown in Figure 2.

4.1 Challenges

To run a unit test with a heterogeneous configuration, ConfAgent needs to be able to control the configuration values at each node. This is trivial in a real distributed setting in which each node is running as a process: we can give each node a separate configuration file. In the setting of unit tests, however, this task becomes significantly more challenging because, to simplify test running, unit tests often create nodes as threads within a single process, and thus there is no way to assign separate configuration files to different nodes.

To address this problem, we observe well-designed applications usually keep track of configuration values in a

Configuration.java — Hadoop Common 3.2.1

```
1   /* Blank constructor */
2   public Configuration() {
3     ConfAgent.newConf(this);
4     …
5   }
6
7   /* Clone constructor */
8   public Configuration(Configuration other) {
9     ConfAgent.cloneConf(other, this);
10    …
11  }
12
13  /* Get the value of name parameter */
14  public String get(String name) {
15    String value = properties.getProperty(name)
16    return value;
17    return ConfAgent.interceptGet(this, name);
18  }
19
20  /* Set the value of the name parameter */
21  public void set(String name, String value) {
22    ConfAgent.interceptSet(this, name, value);
23    …
24  }
```

(a) The configuration class in Hadoop.

NameNode.java — Hadoop-3.2.1 HDFS

```
1   private Configuration conf;
2
3   public static void main() {
4     …
5     /* create a blank config object */
6     Configuration conf = new HdfsConfiguration();
7     NameNode namenode = NameNode(conf);
8     …
9   }
10
11  /* NameNode init function */
12  public NameNode(Configuration conf) {
13    ConfAgent.startInit(this, 'NameNode');
14    this.conf = conf; // save reference
15    /* clone and save the new config reference */
16    this.conf = new Configuration(conf);
17    ConfAgent.refToCloneConf(conf, this.conf);
18    /* initialize this NameNode */
19    …
20    ConfAgent.stopInit();
21  }
```

(b) The NameNode class in HDFS.

TestDeleteBlockPool.java — Hadoop 3.2.1 HDFS

```
1   @Test
2   public void testDfsAdminDeleteBlockPool() {
3     Configuration nnConf = new Configuration();
4     /* create NameNode */
5     NameNode namenode = new NameNode(nnConf);
6     …
7     /* nnConf is used to init DFSAdmin */
8     DFSAdmin admin = new DFSAdmin(nnConf);
9     …
10  }
```

(c) A unit test with configuration sharing.

Figure 2. Instrument the application with ConfAgent APIs (lines with ConfAgent and line 16 in b are added by the developer).

dedicated configuration object (e.g., Configuration class in Hadoop Common, as shown in Figure 2a). Such configuration object usually provides a *get* function to retrieve a certain configuration value and a *set* function to set the value. Therefore, if we can instrument the configuration objects to return different values to different nodes, we can achieve our goal. This approach has the benefit that it only requires instrumentation to a dedicated class. However, the challenge is how to determine which node is calling the *get* function of a particular configuration object. To illustrate the challenge, we present a few attempts we have tried but failed.

Determine caller based on configuration object. Initially, we thought if each node is using one configuration object internally, then our task is trivial: we can annotate the creation of configuration object to connect it to a node. However, when investigating real applications and their unit tests, we find this assumption is almost never true: on the one hand, a unit test often creates a configuration object by itself and then shares the object with different nodes. For example, as shown in Figure 2c, the unit test creates a configuration object, and then uses the object to create a NameNode and a DFSAdmin. In this case, the NameNode and the DFSAdmin are sharing the configuration object. In our experiments, we find configuration object sharing occurs in 96.5%, 99.8%, 100%, 88.5% of the unit tests that involve configuration usage in HDFS, HBase, MapReduce and YARN, respectively. On the other hand, within a node, sometimes the node will create multiple configuration objects, which violates our assumption as well.

Determine caller based on Java object. In the second attempt, we try to tie each Java object to a node, and thus if a Java object calls Configuration.get, we can know which node is making the call. To realize this idea, we first annotate a few objects as roots, which are typically the main object of a node (e.g., DataNode and NameNode in HDFS). Then we apply the following rule: if object A creates object B, then A and B belong to the same node. While we find no correctness problems in this approach, we find it is too invasive: we need to add a "node" field to each object; we need to modify the constructor of each object to pass the "node" field from its creator; we need to do this not only for the target application, but also for any third-party libraries used by the target application. As a result, this approach incurs too much overhead, in terms of both instrumentation effort and runtime CPU and memory usage.

Determine caller based on threads. In the third attempt, we try to implement a simplified version of our second attempt by only keeping track of threads. We annotate the main thread of a node as root, and apply the following rule: if thread A creates thread B, then A and B belong to the same node. Then when a Configuration.get is called, we can retrieve the thread ID of the caller and determine which node is making the call. Compared to the second attempt, this approach is simpler since we can get the thread ID when Configuration.get is called, without tracking the call chain in the middle. However, this approach relies on the assumption that a node's code is only executed by its own threads, and once again, we find the design of unit tests violates our assumption: for testing convenience, it is common for a unit test to directly call nodes' internal functions for different purposes, such as stopping a thread, adding data, checking status, injecting faults, etc. As a result, a node's code may be called by the unit test thread (i.e., main thread), and thus we cannot determine which node is calling Configuration.get.

4.2 ConfAgent's Solution

ConfAgent's solution is based on our first attempt—determine caller based on configuration object. To achieve high accuracy and low instrumentation overhead despite the two challenges (i.e., configuration object sharing across different nodes and multiple configuration objects within a node), ConfAgent incorporates a number of heuristics, based on our study about how configuration objects are created and used in both the unit tests and the applications.

Observation 1: the number of types of nodes is small. As discussed previously, for all methods, we need to define certain "root" classes or objects to separate different nodes at runtime. Fortunately we find this is a simple task: all the applications we investigated have a well-defined node class for each type of node, e.g., NameNode and DataNode in HDFS, ResourceManager and NodeManager in YARN. The number of types of nodes is small, which means manual annotation is feasible. Table 1 records node types we picked in our work.

Observation 2: flow of configuration objects. We observe the information of configuration objects can flow in three ways, which provide hints about how to track them.

- Observation 2.1: create a new blank configuration object. Both the unit test itself and a node may create new configuration objects. We observe a node usually creates its configuration objects in an initialization function, typically the constructor function or an "init" like function in the node class. This means we can annotate the initialization function to map a configuration object to its node. Note that first, sometimes the object is created by a function called by the initialization function, instead of by the initialization function itself; and second, sometimes a node may create multiple configuration objects, usually for its subcomponents. To capture such relationship, ConfAgent adds the following rule: if a configuration object is created at time t_1 on thread A, and thread A executes the initialization function of a node between t_2 and t_3, and $t_2 < t_1 < t_3$, then the configuration object belongs to the particular node.
- Observation 2.2: clone a configuration object. Both the unit test and the application may clone a configuration object by creating an object with a constructor which copies values from an existing object. We observe the original object and the cloned object usually belong to the same entity (i.e., a node or the unit test itself).
- Observation 2.3: create a new reference to a configuration object. This will not create a new object, and thus will not affect the mapping from configuration objects to nodes, except the following case: we observe a node's initialization function often takes a configuration object as an argument and assigns it to an internal reference (line 13 in Figure 2b). In a real distributed setting, the main function of the node class will create the configuration object and use it to create the node class (line 6-7 in Figure 2b), but in the unit test, the unit test itself will replace the main function and may share the configuration object with many nodes (lines 3-8 in Figure 2c). To solve this problem, we require the user to replace such configuration object reference in the initialization function as a clone.

Observation 3: there are exceptions to previous observations, which create the main source of false positive. Exceptions to previous observations may cause ConfAgent to fail to assign proper values to different configuration objects and cause false positives. In particular, if ConfAgent assigns different values to configuration objects within the same node, it may cause errors but this should not happen in a real setting. Although the total number of exceptions is small compared to the total number of unit tests, since the number of unsafe heterogeneous configurations is small as well, the exceptions can be a main source of false positives. ConfAgent has no perfect solution to this problems. Instead, it tries to identify such cases: ConfAgent tries to map each configuration object to either a node or the unit test itself; if eventually it finds a configuration object is mapped to no entity, then ConfAgent will report a warning to the developer.

Based on such observations, ConfAgent works as follows: first, the developer needs to annotate the node initialization and configuration object creation with the APIs provided by ConfAgent (see Section 4.3). Then at runtime, ConfAgent follows a number of rules to determine the mapping from configuration object to node, as discussed above. Here we summarize all rules:

Rule 1.1: Configuration object creation (Observation 2.1). If a configuration object is created at time t_1 on thread A, and thread A executes the constructor of a node between t_2 and t_3,

System	Types of Nodes
HDFS	NameNode, DataNode, SecondaryNameNode, JournalNode, Balancer, Mover
HBase	HMaster, HRegionServer, ThriftServer, RESTServer
Yarn	ResourceManager, NodeManager, ApplicationHistoryServer
MapReduce	MapTask, ReduceTask, JobHistoryServer

Table 1. The types of nodes we investigated in different applications.

and $t_2 < t_1 < t_3$, then the configuration object belongs to the particular node.

Rule 1.2: Configuration object creation. If a configuration object is created when no node has started, then this object belongs to "unit test".

Rule 2: Configuration object reference (Observation 2.3). If the developer replaces a configuration object reference with a clone during the initialization, then the object to be cloned belongs to "unit test", and the cloned object belongs to the node which executes the initialization function.

Rule 3: Configuration object clone (Observation 2.2). If a configuration object is cloned from another one not by Rule 2, then these two objects belong to the same entity.

With these rules, ideally ConfAgent should be able to map each configuration object to either "unit test" or a node: if this is not true, ConfAgent will report a warning to the developer. The current implementation of ZebraConf skips these unit tests with unidentifiable configuration objects since they may generate false positives. Our evaluation (Table 5) shows that for three out of the four target applications, less than 5% of the tests are skipped; for the remaining one, about 10% of the tests are skipped. Such numbers indicate that these rules accurately cover a high percentage of unit tests.

4.3 ConfAgent APIs and implementation

To implement all the rules, ConfAgent provides a few APIs for the developer to annotate the source code of the target application:

- *startInit(node, nodeType)* and *stopInit()*. The developer should use these two APIs to annotate the start and the end of the initialization function (e.g., lines 13 and 20 in Figure 2b), which are to implement Rule 1.1.
- *newConf(confObj)*, *cloneConf(origConfObj, newConfObj)*, and *refToCloneConf(origConfObj, newConfObj)*. These three APIs are to track configuration objects, which are used to implement all the rules mentioned above. The user can annotate the constructor of the configuration class with *newConf* or *cloneConf*, depending on whether the constructor copies values from another configuration object (e.g., lines 3 and 9 in Figure 2a). When the user replaces a configuration object reference with a clone (Rule 2), she should annotate it with *refToCloneConf* (e.g., lines 16-17 in Figure 2b).
- *interceptGet(confObj, paraName)* and *interceptSet(confObj, paraName, paraValue)*. These two APIs are to intercept the get and set functions of configuration objects, which are to implement heterogeneous configuration. The user can place these two APIs in the get and set functions of the configuration class (e.g., lines 17 and 22 in Figure 2a).

To implement these APIs, ConfAgent maintains a *nodeTable* of [nodeID, nodeType, nodeIndex, confObjIDs[], parentConfObjID]: nodeID is the hashCode of the corresponding node; nodeType is the type of the node (e.g., NameNode, DataNode, etc); nodeIndex indicates the corresponding node is the nodeIndex-th node of nodeType, which is to address the problem that hashCode may not be consistent across multiple runs (i.e., TestGenerator will assign a value to the first node, instead of node with hashCode 234); confObjIDs array records the hashCode of all configuration objects belonging to this node; parentConfObjID records the hashCode of the configuration object passed as the argument to the initialization function, if any. Furthermore, ConfAgent maintains a *unitTestConfObjIDs* list to maintain configuration objects mapped to "unit test", an *uncertainConfObjIDs* list to maintain configuration objects that cannot be mapped to anywhere, a *parent-child<childConfObjID, parentConfObjID>* map to keep track of all cloning relationships, and a *threadContext<ThreadID, nodeID>* map to keep track whether an initialization function is executed on a thread.

When *startInit(node, nodeType)* is called, ConfAgent will put a new entry in *nodeTable* (confObjIDs is empty and parentConfObjID is null). It will further put current thread ID and node ID in *threadContext*. When *stopInit* is called, ConfAgent will remove current thread ID from *threadContext*.

When *newConf*, *cloneConf*, or *refToCloneConf* is called, ConfAgent will update the information based on the rules mentioned previously:

- When *newConf(confObj)* is called, if no node has initialized yet, ConfAgent will put the confObj in *unitTestConfObjIDs* (Rule 1.2); if *threadContext* has a pair of <ThreadID, nodeID> for the current thread, ConfAgent will put the confObj into *nodeTable* (Rule 1.1); otherwise, ConfAgent will put the confObj in *uncertainConfObjIDs*.
- When *cloneConf(origConfObj, newConfObj)* is called, ConfAgent will search if either origConfObj or newConfObj already belongs to a node (i.e., in *nodeTable*) or the unit test (i.e., in *unitTestConfObjIDs*): if so, ConfAgent will put the other confObj in the same group (Rule 3); otherwise,

	#Unit Tests	#Parameters
HDFS	6445	579 [16, 17] + 336 [8]
HBase	4985	206 [34] + 336 [8]
YARN	4806	465 [33] + 336 [8]
MapReduce	1423	210 [26] + 336 [8]
Hadoop Tools	1518	N/A

Table 2. Statistics for different applications. Hadoop Tools provide a number of tools to support other applications, but do not have their own parameters. All other applications share the Hadoop Commons library, which has 336 parameters.

ConfAgent will put both configuration objects in *uncertainConfObjIDs*. In either case, ConfAgent will put the pair in *parent-child* map.

- When *refToCloneConf(origConfObj, newConfObj)* is called, ConfAgent will put newConfObj in *nodeTable* with the nodeID retrieved from threadContext, and put origConfObj in *unitTestConfObjIDs* (Rule 2). Furthermore, ConfAgent will recursively search origConfObj's parent in *parent-child* map to move them from *uncertainConfObjIDs* to *unitTestConfObjIDs* (Rule 3).

At the end of a unit test, if *uncertainConfObjIDs* is not empty, ConfAgent will skip the test.

When *interceptGet(confObj, paraName)* is called, ConfAgent will first search whether confObj is in *nodeTable*: if so, ConfAgent can retrieve its corresponding nodeType and nodeIndex and check whether TestGenerator has assigned a particular value to <nodeType, nodeIndex, paraName>, in which case *interceptGet* will return the assigned value; if confObj is not in *nodeTable* or if TestGenerator has not assigned a particular value, ConfAgent will return the original value in confObj.

ConfAgent utilizes *interceptSet* to solve the following problem: sometimes the unit test will create a configuration object with empty values, then create a node with this configuration object, expecting the node to fill the empty values, and later retrieve these values. Since we replace the reference with a clone, the unit test won't be able to get the correct values after node initialiation. To solve this problem, ConfAgent adds the following logic to *interceptSet(confObj, paraName, paraValue)*: if confObj belongs to a node in *nodeTable* and the parentConfObjID of the particular node is not empty, then ConfAgent will update the corresponding value of the parent confObj as well.

5 Design of TestGenerator

TestGenerator is responsible for generating all the test instances. In the general case, a test instance is represented by a tuple of a unit test and a heterogeneous configuration $HeterConf(F_1, F_2, ..., F_n)$. Table 2 shows the number of unit tests and parameters in different applications. As one can imagine, enumerating the combination of all possible heterogeneous configurations and all unit tests will generate too many test instances. To exacerbate the problem, we observe many unit tests, in particular the ones we are interested in, can take long (e.g. several minutes), because they need to wait for a cluster to be set up. To alleviate this problem, we introduce a number of strategies and techniques:

Test each parameter independently. We assume whether a configuration parameter is unsafe when set in a heterogeneous manner will not depend on the values of other parameters. This assumption allows us to test each individual parameter rather than testing their combinations, which greatly reduces the number of test instances. Of course this assumption may not always hold, which will introduce false negatives. We plan to take the relationship of different parameters into consideration in the future, by relying on parameter dependency analysis [4].

With this strategy, TestGenerator converts the representation of a test instance into a tuple of 1) a unit test, 2) the name of the parameter to test, and 3) the parameter value at each node. All other parameters will use the original values in the particular unit test.

Select parameter values to test. For boolean parameters, this task is trivial since we only need to test true and false values. For other types of parameters, we manually select a few parameter values which we believe representative based on the document of the target application: for numerical values, we select one that is much larger than the default value, one that is much smaller, and values that have specific meanings (e.g., 0 sometimes means this feature is disabled). For string values, we select a few based on the meaning of the parameter. For example, for a string to determine encoder type, we select the possible encoder names.

Select representative value assignment. If a unit test contains n nodes and we need to test a parameter with two different values v_1 and v_2, then there are 2^n ways to assign values to nodes. To reduce this number, we select a few representative assignment strategies. We first divide nodes into groups based on their types, and then test each group with a heterogeneous configuration (i.e., all other groups get v_1): for the particular group, we test three assignments [v_2, v_1, v_2, ...], [v_1, v_2, v_1, ...], and [v_2, v_2, v_2, ...]. In our experiments, such settings can find both problems happening across different groups and problems happening within the same group.

Pre-run unit tests. TestGenerator pre-runs all unit tests once to filter useless unit tests and useless combination of tests and parameters.

First, not all nodes in all unit tests will use all parameters. If we assign a parameter value to a node not using the parameter, then of course we are wasting our time.

This fact provides an opportunity for us to trim the number of tests to run. To exploit this opportunity, during the pre-run

TestGenerator records which node is using which parameter in each unit test. When generating test instances, TestGenerator applies the following rule: for a unit test with nodes of type A and a parameter p, TestGenerator will only generate test instances to test p on nodes of type A if these nodes actually use p in the pre-run. For example, in HDFS, TestGenerator will not test dfs.datanode.balance.bandwidthPerSec on NameNode because NameNode never uses this parameter.

A technical challenge of this technique is how to determine whether a node "uses" a parameter. In our current implementation, we define "use" as reading a parameter, which is easy to implement but may be loose since a node may read a parameter during initialization and never use it later. We will explore whether programming analysis can further improve the accuracy of identifying the usage of a certain parameter in the future.

As shown in our evaluation (Table 5), pre-running unit tests and filtering useless tuples allows us to reduce the number of unit tests to run by up to three orders of magnitude.

Pooled testing. To further reduce testing time, we observe that most configuration parameters are safe in a heterogeneous setting. This motivates us to use the classic divide and conquer approach: instead of running one unit test to test one parameter, we can run the unit test to test multiple parameters together; if the unit test does not report any errors, then all these parameters are safe; otherwise, we divide these parameters into two groups and test each group recursively, till we can locate all unsafe parameters. In our evaluation, we use a group size of 20.

In order for this approach to be effective, we expect most groups should not return errors, which allows to test multiple parameters with one run. In our testing, however, we find the efficiency of this approach is hampered by a small number of unsafe parameters that fail almost every unit test. Examples include those encryption or compression related parameters, which are used in most unit tests. To solve this problem, if TestGenerator finds a parameter has failed many unit tests, TestGenerator will mark it as unsafe and avoid to test this parameter in the future unit tests.

Test in parallel. Unit tests are independent from each other, which provides a natural opportunity to run unit tests in parallel. In our experiments, we run unit tests on a cluster of machines to speed up them.

6 Design of TestRunner

TestRunner is responsible for running a test instance generated by TestGenerator. Based on Definition 3.1, TestRunner will test both the heterogeneous configuration generated by TestGenerator and all corresponding homogeneous configurations: if the former reports an error and the latter does not, then TestRunner will report an unsafe heterogeneous configuration.

TestRunner's task is complicated by non-deterministic errors in unit tests. For example, if the heterogeneous test has a probability to fail but does not fail in one test, then we may miss an unsafe heterogeneous configuration parameter (i.e., false negative); if one of the homogeneous configurations has a probability to fail but does not fail in one test, then we may report an unsafe heterogeneous parameter incorrectly (i.e. false positive).

Our principle is to minimize the false positives during one test: if TestRunner finds the heterogeneous configuration fails but all homogeneous configurations succeed, TestRunner will further run both the heterogeneous configuration and the homogeneous configurations multiple times and use hypothesis testing to test whether their probability of failure differ significantly (we use a significance level of 1 - 99.99% in our testing).

However, this approach does not address the false negatives since heterogeneous configuration not failing and homogeneous configuration failing will be directly regarded as no problem. If the user wants to reduce false negatives, she needs to run the tests multiple times, which is not ideal but is the standard solution to most non-deterministic errors. In our experiments, we find false positives caused by nondeterministic error are common but false negatives are uncommon since a parameter is usually tested by multiple unit tests, which naturally reduces false negatives.

7 Experimental Evaluation

We implement ConfAgent with 622 lines of Java code, TestRunner with 450 lines of Java Code and 121 lines of shell script, and TestGenerators with 133 lines of Java code and 111 lines of shell script. We also have 319 lines of shell script to run tests on Docker.

Our evaluation tries to answer two questions:

- How many unsafe heterogeneous configuration parameters can ZebraConf find in real-world applications? (Section 7.1)
- How can the individual techniques of ZebraConf help to improve its accuracy and reduce its running time? (Section 7.2)

To answer these questions, we have applied ZebraConf to HDFS, HBase, MapReduce and YARN. We manually analyze all the reported problems to understand whether they are true problems or false positives.

Testbed. We run all experiments on CloudLab [6]. Each machine is equipped with Two Intel Xeon 10-core CPUs, 192GB DRAM, 480 GB SATA SSD (where we run experiments) and 1 TB SAS HDs. We use 100 physical machines and allocate 20 Docker containers on each physical machine to run unit tests in parallel.

Parameter	App:Node
dfs.block.access.token.enable	HDFS:NameNode
dfs.bytes-per-checksum	HDFS:NameNode, HDFS:DataNode
dfs.checksum.type	HDFS:NameNode, HDFS:DataNode
dfs.client.block.write.replace-datanode-on-failure.enable	HDFS:NameNode
dfs.client.socket-timeout	HDFS:DataNode, HBase:HMaster, HBase:HRegionServer
dfs.datanode.balance.bandwidthPerSec	HDFS:DataNode
dfs.datanode.balance.max.concurrent.moves	HDFS:DataNode
dfs.data.transfer.protection	HDFS:DataNode, HDFS:Balancer
dfs.encrypt.data.transfer	HDFS:NameNode, HDFS:DataNode
dfs.ha.tail-edits.in-progress	HDFS:NameNode, HDFS:JournalNode
dfs.heartbeat.interval	HDFS:NameNode, HDFS:DataNode
dfs.namenode.fs-limits.max-component-length	HDFS:NameNode
dfs.namenode.fs-limits.max-directory-items	HDFS:NameNode
dfs.namenode.snapshotdiff.allow.snap-root-descendant	HDFS:NameNode
dfs.namenode.upgrade.domain.factor	HDFS:Balancer, HDFS:NameNode
hadoop.rpc.protection	HDFS:NameNode, HBase:HRegionServer, YARN:NodeManager,...
hbase.regionserver.thrift.compact	HBase:ThriftServer
hbase.regionserver.thrift.framed	HBase:ThriftServer
ipc.client.rpc-timeout.ms	HDFS:DataNode, HDFS:NameNode, HBase:HMaster
mapreduce.fileoutputcommitter.algorithm.version	MR:MapTask, MR:ReduceTask
mapreduce.job.encrypted-intermediate-data	MR:MapTask, MR:ReduceTask
mapreduce.job.maps	MR:ReduceTask
mapreduce.job.reduces	MR:MapTask, MR:ReduceTask
mapreduce.map.output.compress	MR:MapTask
mapreduce.shuffle.ssl.enabled	YARN:NodeManager
yarn.timeline-service.enabled	YARN:ResourceManager
yarn.node-labels.configuration-type	YARN:ResourceManager

Table 3. Unsafe heterogenous parameters found by ZebraConf.

7.1 Unsafe heterogeneous configuration parameters in real-world applications

ZebraConf reports a total number of 58 heterogeneous unsafe parameters in the four target applications, and our manual analysis reveals 27 of them are true problems. In the remaining ones, 4 of them happen to trigger errors unrelated to heterogeneous and 27 of them are false positives. We list all true problems in Table 3.

We categorize the true problems as follows and we discuss them in both long-term heterogeneous configuration and short-term heterogeneous configuration (i.e., partial reboot in a homogeneous system).

- Compression, encryption, authentication, or transport protocols related parameters. These parameters obviously should not be configured in a heterogeneous manner, either long-term or short-term. And we don't have a particular solution to these parameters.
- Heartbeat related parameters. If a heartbeat sender has a large interval value but the receiver has a small value, the sender may not send heartbeats in time, so that the receiver may consider the sender as dead. While there is no good reason to use heterogeneous heartbeat intervals in the long term, we observe this may happen in a short term due to the demand to re-configure such values at runtime. For example, since 2.9.0 version, HDFS started to support re-configuring the parameter *dfs.heartbeat.interval* at runtime with its reconfiguration interface *hdfs dfsadmin -reconfig namenode* [11] . Such online re-configuration will create a short-term unsafe heterogeneous configuration. We propose the following work-around: if the administrator needs to decrease the heartbeat interval, then she should change the value at the heartbeat sender first, and then change the value at the receiver; if the administrator needs to increase the interval, she should change it at the receiver first and then at the sender. This strategy ensures that the sender interval is always less than or equal to the receiver interval, so that the receiver will not miss heartbeats. However, this work-around may not always work, since sometimes a node can act as both the sender and the receiver.
- Max limit related parameters. This group can encounter problems if we re-configure a node's max limit to be smaller, while the state of the node already exceeds the smaller limit. We don't have a solution to this problem and the administrator should simply not do this. Increasing the limit is fine in our experiments.
- Number of mappers and reduces. If these parameters are inconsistent at different nodes, the nodes will have problems

retrieving data. Within a MapReduce job, these parameters should not be configured in a heterogeneous manner.

- Others. This group contains some interesting parameters that we do not expect to be unsafe. We provide details about some of them as follows:

dfs.datanode.balance.bandwidthPerSec This parameter is used to specify the maximum amount of bandwidth that each HDFS Datanode can utilize for the balancing purpose. From HDFS 0.20, developers made this parameter online reconfigurable with a new dfsadmin command, because admins found the optimal value is never known at system booting time. Either setting bandwidthPerSec too low or too high may bring the cluster into a 'maintenance window', which is expensive and a serious maintenance problem for large clusters. Making this parameter reconfigurable at runtime helps to avoid or alleviate this issue.

"*The optimal value of the bandwidthPerSec parameter is not always (almost never) known at the time of cluster startup*" – HDFS-2171 [12]

However, when setting this parameter in a heterogeneous manner, we observe the following problem: a DataNode with a high bandwidth limit may send many packets to a DataNode with a low limit so that the latter may run out of its quota. In this case, the latter will throttle its network traffic, which is expected. However, such throttling will block the heartbeat traffic to the NameNode. If this phenomenon lasts for a while, the NameNode will mark the DataNode as dead. To solve this problem, we propose that each node should reserve a small fraction of bandwidth for critical traffic like heartbeats.

dfs.datanode.balance.max.concurrent.moves. ZebraConf reports this parameter as heterogeneous unsafe because it fails the test TestBalancer#testUnknownDatanodeSimple. ZebraConf reports an issue when this parameter is configured to be 1 on DataNodes and 50 on HDFS Balancer, which is a tool for administrators to balance data in the cluster. The default value for this parameter is 50, which allows 50 threads on a DataNode to transfer blocks for balancing purpose.

This unit test reports timeout (100s) when testing with the aforementioned heterogeneous configuration setting. We check the average balancing time for (DataNode:50, Balancer:50), (DataNode:1, Balancer:1), and (DataNode:1, Balancer:50): the time for (DataNode:50, Balancer:50) is 14 seconds, for (DataNode:1, Balancer:1) is 16.7 seconds, but the time for (DataNode:1, Balancer:50) is 154 seconds. While it is reasonable for (DataNode:50, Balancer:50) to be faster than (DataNode:1, Balancer:1), it is unclear why (DataNode:1, Balancer:50) is significantly slower than (DataNode:1, Balancer:1), so we add more debugging output in the corresponding code.

The log shows that in the setting (DataNode:1, Balancer:50), because Balancer is unaware of the 1 thread capacity on DataNodes, it will still send block transfer requests to DataNodes concurrently. However, DataNodes will decline requests when its thread is already performing block transfer for balancing. When the request is declined, the corresponding Balancer dispatcher thread triggers a congestion control mechanism, which will sleep for 1100ms before it retries. Since the DataNodes usually can finish its rebalancing task within 1100ms, such congestion control incurs extra delay to the whole procedure.

One may wonder why we want to set this parameter differently on Balancer and on DataNodes in the first place. Indeed, if all DataNodes have the same value, there is no good reason for the Balancer to use a different value. However, if different DataNodes have different values, then it is inevitable that the Balancer's configuration will be different from some of the DataNodes. Because of the reported problem, it seems not a good idea for the Balancer to use one value for this parameter: instead, it should retrieve this value from different DataNodes and accordingly send different number of tasks to different DataNodes. We observe the community is already discussing this solution [14].

For curiosity, we take a look at the history of this parameter to see if there is a demand for heterogeneous configuration: before HDFS 2.5.0, the maximum number of threads that a DataNode allows to use for data balancing is 5. Since many people argue that this value is too small (balancing is slow) and this parameter is hardware specific [10, 13], HDFS developers made it configurable since HDFS 2.5.0. Later, people are interested in performing online reconfiguration for this parameter, and the developers did it since HDFS 2.8.0 [15]. These discussions show that there is a demand to configure this parameter in a heterogeneous manner.

dfs.namenode.upgrade.domain.factor. This parameter is effective when block placement policy is set to BlockPlacementPolicyWithUpgradeDomain. Upgrade domain is a feature introduced from HDFS 2.4.0 to support rolling upgrade, which upgrades a subset of DataNodes at a time. To minimize the chance of data unavailability, such rolling upgrade should affect at most one replica of a data block at a time. To achieve this property, HDFS allows the administrator to divide DataNodes into groups, called upgrade domains, and then will ensure the replicas of a data block are placed into different upgrade domains.

In TestBalancer#testUpgradeDomainPolicyAfterBalance, which tests whether data rebalancing will still honor domain-aware block placement policy, ZebraConf catches a heterogeneous configuration issue when Balancer and NameNode are configured with different numbers of upgrade domains. The rebalancing task will never finish because some block transferring requests will always be declined by NameNode, which identifies the block transfer as an action that result in a violation of the domain-aware block placement policy being used.

System	Instrumented LOC
HDFS	30 + 6
HBase	20 + 7
Yarn	15 + 6
MapReduce	15 + 6
Hadoop Common	6 + 0

Table 4. LOC of modification to apply ZebraConf to different applications. The first number shows the lines related to modify the node classes and the second number shows lines related to modify the configuration class

Similar as the previous problem, if different NameNodes (HDFS started to support multiple NameNodes since version 0.23) have the same number of UpgradeDomains, there is no good reason for the Balancer to use a different value. If different NameNodes have different number of UpgradeDomains, however, it is inevitable for the Balancer's configuration to be different from some HDFS NameNodes. A possible solution for this issue is to let Balancer fetch the value of domain factor from the corresponding NameNode, instead of reading from its local configuration file.

Causes of false positives We summarize the top causes of false positives:

- Restrictive assertions in unit tests. A number of unit tests have strict assertions throughout the tests. Some of such assertions will check the configuration values, and thus changing these values will cause the unit tests to fail directly.
- Exception handling. Some unit tests expect an exception to occur during the tests and thus will try to catch such expected exceptions. If a new configuration value will cause the same type of exception, the unit tests will consider it as normal and report no error. Therefore, if a homogeneous configuration should fail but is reported as no error, it may incur false positives.
- Inappropriate timeout values. Some unit tests have an internal timeout, indicating if data or a node is not available after a given amount of time, the unit test fails. Such timeout values are tuned based on the configuration values, and thus if we change these values, the timeout values may not be appropriate any more.

7.2 Effects of individual techniques

Effort to instrument the applications. As discussed in Section 4.3, to use ZebraConf, the user needs to use ConfAgent's APIs to instrument two types of class files: one is the node class, and the other is the configuration class. Table 4 shows the lines of code we need to instrument each application. As one can see, ZebraConf does not require a large effort.

Effects of different techniques. Table 5 presents the effects of individual techniques incorporated by ZebraConf.

The first row computes the number of unit tests ZebraConf would run assuming the user has the same level of expertise as us but does not further analyze those unit tests. In particular, it assumes the user tests each parameter independently, select parameter values in the same way as us, and select value assignment in the same way as us (Section 5). It also assumes the user knows which types of nodes the corresponding application include, so she will not test nodes or parameters not included in the application. For example, for HDFS, she will not test RegionServer and related parameters, which belong to HBase; for HBase, however, she will test HDFS NameNode or DataNode and related parameters, because HBase depends on HDFS. As one can see in the table, with such a naive approach, ZebraConf needs to run a large number of unit tests.

The second row computes the number of unit tests after we pre-run the unit tests to filter useless ones. As one can see, such profiling has significantly reduced the number of tests to run. By looking at the test information, we observe 1) many unit tests, which are designed to test individual data structures, do not even start any nodes, and thus are completely filtered by this step; 2) almost none of the unit tests use all parameters; 3) even for unit tests that use a certain parameter, in many cases the parameter is only used by a subset of nodes. These reasons combined allow us to reduce the number of unit tests to run by up to three orders by magnitude.

The third row computes the number of unit tests after we remove those with uncertain configuration objects (Section 4). As one can see, most of the test instances will not meet uncertain configuration objects, which have confirmed the accuracy of ZebraConf's approach to map configuration objects to nodes. However, note that although the percentage of tests with uncertainties is small, if we don't remove them, they will generate a high false positive, because the percentage of unsafe parameters is small as well.

The last row records the number of unit tests ZebraConf actually runs after applying pooled testing, including the number of pooled tests and the number of individual tests if a pooled test fails. As one can see, pooled testing further reduces the number of tests ZebraConf needs to run.

These techniques combined reduce the number of tests to run by two to three orders of magnitude. With their help, ZebraConf is able to finish all tests on a cluster of 100 machines within 41 hours. While these numbers are not small, they at least become affordable, considering we are not supposed to run such tests frequently.

Effects of hypothesis testing. In our tests, ZebraConf reports 2098 tests as failed in the first place (i.e. the heterogeneous configuration fails but all corresponding homogeneous tests succeed), and hypothesis testing filters 714 of them as false positives. As these numbers show, nondeterministic errors do happen quite frequently, which has confirmed the necessity of the probabilistic approach.

	HDFS	HBase	YARN	MapReduce	Hadoop-Tools
Original	387,499,008	557,761,680	705,346,824	284,486,160	373,850,400
After pre-running unit tests	10,404,952	6,145,374	668,020	482,272	356,016
After removing uncertainty	10,242,886	6,033,174	640,338	430,800	346,588
After pooled testing	1,968,218	1,438,929	312,726	104,588	89,744

Table 5. The number of test instances generated by different methods.

8 Conclusion

In this work, we have built ZebraConf, a framework to utilize existing unit tests to test whether a configuration parameter is safe to be configured in a heterogeneous manner. ZebraConf incorporates APIs to instrument the target application with minimal effort and a number of strategies to reduce the number of tests to run. Our evaluation on four popular open-source applications find 27 unsafe heterogeneous configuration parameters, which shows that an administrator should be careful when applying heterogeneous configurations.

References

[1] Apache HBASE. http://hbase.apache.org/.

[2] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, volume 10, pages 1–14, 2010.

[3] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. Autoconfig: Automatic configuration tuning for distributed message systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 29–40. IEEE, 2018.

[4] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[5] Dazhao Cheng, Jia Rao, Yanfei Guo, Changjun Jiang, and Xiaobo Zhou. Improving performance of heterogeneous mapreduce clusters with adaptive task tuning. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):774–786, 2016.

[6] CloudLab. https://www.cloudlab.us.

[7] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[8] Hadoop-3.2.1 hadoop common configuration file. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/core-default.xml.

[9] Hdfs. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[10] Increase balancer defaults further. https://issues.apache.org/jira/browse/HDFS-14675.

[11] Support reconfiguring dfs.heartbeat.interval and dfs.namenode.heartbeat.recheck-interval without nn restart. https://issues.apache.org/jira/browse/HDFS-1477.

[12] Changes to balancer bandwidth should not require datanode restart. https://issues.apache.org/jira/browse/HDFS-2171.

[13] Configure the maximum threads allowed for balancing on datanodes. https://issues.apache.org/jira/browse/HDFS-6595.

[14] Allow different values for dfs.datanode.balance.max.concurrent.moves per datanode. https://issues.apache.org/jira/browse/HDFS-7466.

[15] Support reconfiguring dfs.datanode.balance.max.concurrent.moves without dn restart. https://issues.apache.org/jira/browse/HDFS-9214.

[16] Hadoop-3.2.1 hdfs configuration file. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[17] Hadoop-3.2.1 hdfs-rbf configuration file. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs-rbf/hdfs-rbf-default.xml.

[18] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[19] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. ConfValley: A systematic configuration validation framework for cloud services. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, pages 19:1–19:16, New York, NY, USA, April 2015. ACM.

[20] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. Association for Computing Machinery.

[21] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC 14, pages 165 – 176, New York, NY, USA, 2014. Association for Computing Machinery.

[22] Wenhao Lyu, Youyou Lu, Jiwu Shu, and Wei Zhao. Sapphire: Automatic configuration recommendation for distributed storage systems. *arXiv preprint arXiv:2007.03220*, 2020.

[23] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 189–203. USENIX Association, July 2020.

[24] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware 17, pages 28–40, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online reconfiguration of clustered nosql databases for time-varying workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 223–240, Renton, WA, July 2019. USENIX Association.

[26] Hadoop-3.2.1 mapreduce configuration file. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml.

[27] Mapreduce. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[28] Ariel Shemaiah Rabkin. *Using program analysis to reduce misconfiguration in open source systems software*. PhD thesis, UC Berkeley, 2012.

[29] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 18, pages 154 – 168, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. Pracextractor: Extracting configuration good practices from manuals to detect server misconfigurations. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 265–280. USENIX Association, July 2020.

[31] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 619–634, 2016.

[32] Yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[33] Hadoop-3.2.1 yarn configuration file. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-common/yarn-default.xml.

[34] Hbase-2.2.4 configuration file. https://hbase.apache.org/book.html#hbase_default_configurations.

[35] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *2011 USENIX Annual Technical Conference (USENIX ATC2011)*, 2011.

[36] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 415 – 432, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 687–700, 2014.

[38] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC 17, pages 338 – 350, New York, NY, USA, 2017. Association for Computing Machinery.