





Automatic Debugging of Design Faults in MapReduce Applications

Jesús Morán , Antonia Bertolino , Claudio de la Riva , and Javier Tuya , *Member, IEEE*

Abstract—Among the current technologies to analyse large data, the MapReduce processing model stands out in Big Data. MapReduce is implemented in frameworks such as Hadoop, Spark or Flink that are able to manage the program executions according to the resources available at runtime. The developer should design the program in order to support all possible non-deterministic executions. However, the program may fail due to a design fault. Debugging these kinds of faults is difficult because the data are executed non-deterministically in parallel and the fault is not caused directly by the code, but by its design. This paper presents a framework called MRDebug which includes two debugging techniques focused on the MapReduce design faults. A spectrum-based fault localization technique locates the root cause of these faults analysing several executions of the test case, and a Delta Debugging technique isolates the data relevant to trigger the failure. An empirical evaluation with 13 programs shows that MRDebug is effective in debugging the faults, especially when the localization is done with the reduced data. In summary, MRDebug automatically provides valuable information to understand MapReduce design faults as it helps locate their root cause and obtains a minimal data that triggers the failure.

Index Terms—Debugging aids, testing and debugging.

I. INTRODUCTION

THE data generated during recent years has grown exponentially, and consequently several challenges arise to store, transform and analyse this information. Traditional technologies are not able to handle huge amounts of data in a scalable way and the *Big Data* paradigm [1] has emerged to tackle these challenges through novel technologies. Most of the de facto standard *Big Data* frameworks are based on the *MapReduce* processing model, among others *Hadoop MapReduce* [2], *Spark* [3] or *Flink* [4]. One study in Microsoft [5] indicates that at least 34% of data scientists (*Big Data* and non *Big Data*) use frameworks based on

MapReduce. The *MapReduce* processing model [6] divides one big problem into several small subproblems that are distributed in a large cluster, and the execution is automatically managed by a framework. One program can be executed in different ways because the framework allocates the resources not only based on the program design, but also based on the resources available at runtime. For example, the framework could execute the same program with more or less parallelism, or re-execute part of the program in case of infrastructure failures. If that same program code executes the same dataset several times and sometimes succeeds but other times fails, depending on how the framework manages the execution, then this program is incorrectly designed. The developer should design the program so that it behaves correctly in all possible executions, because the framework will manage the execution depending on, among others, the runtime resources. To avoid these failures, the program should satisfy the semantics required by the *MapReduce* processing model like idempotency, and commutativity/associativity, among others.

The quality of the *MapReduce* programs becomes crucial, especially in those used in health (e.g., DNA alignment [7]), security (e.g., ballistics [8]), or other critical sectors. An analysis performed in the *MapReduce* clusters at Yahoo! indicates that around 3% of the programs do not finish the execution [9]. Another study places this percentage between 1.38% and 33.11% [10]. Concerning the characterization of faults, another study of 507 programs reveals at least five different kinds of faults are caused by an erroneous design [11]. Other works identify and categorize further more faults that are caused by an incorrect design of the *MapReduce* applications [12], [13].

In our previous work we have devised a testing technique that can detect these design faults automatically when the same test case executed several times does not yield similar outputs [14]. According to an empirical study [15], sometimes the developers erroneously think that these design faults are caused by the framework malfunction and report it, but in fact the fault lies in the program design and not in the underlying framework. Not only are design faults difficult to detect during testing, they are also difficult to debug as the execution generally exhibits parallelism and timing issues, among others. Sometimes the design faults are masked, and other times manifest themselves in a non-deterministic way due to the distributed execution of *MapReduce*. According to a large-scale study conducted by Bagherzadeh et al. [16], *MapReduce* and debugging are among the three most requested topics in Stack Overflow about *Big Data*. In the case of debugging, some studies suggest that developers could benefit by the integration of several debugging techniques [17].

Manuscript received 31 March 2023; revised 8 September 2023; accepted 13 February 2024. Date of publication 26 February 2024; date of current version 19 April 2024. This work was supported in part by the project PID2019-105455GB-C32 under Grant MCIN/AEI/10.13039/501100011033 (Spain), in part by the project PID2022-137646OB-C32 under Grant MCIN/AEI/10.13039/501100011033/FEDER, UE, and in part by the project RDS_2022-2024_2.1_Progetto_CYBER under Grant MASE/PTR_22_24_INT_2_1 (Italy). Recommended for acceptance by N. Nagappan. (*Corresponding author: Jesús Morán.*)

Jesús Morán, Claudio de la Riva, and Javier Tuya are with the Computer Science Department, University of Oviedo, 33203 Oviedo, Spain (e-mail: moranjesus@uniovi.es; claudio@uniovi.es; tuya@uniovi.es).

Antonia Bertolino is with the ISTI-CNR, 56124 Pisa, Italy (e-mail: antonia.bertolino@isti.cnr.it).

Digital Object Identifier 10.1109/TSE.2024.3369766

This paper proposes, integrates and implements in a framework called MRDebug two debugging techniques aimed to help developers during the understanding of the *MapReduce* design faults. Indeed, identifying the causes and locations of *MapReduce* design faults can be complex due to the many different ways in which the *Big Data* framework can execute the program and the size of input data. For this reason, we propose a specific debugging framework that addresses both i) the automatic localization of the root cause of the fault using a fault localization technique, and ii) the automatic isolation of the data that trigger the failure using an input reduction technique. These two debugging techniques focus on different aspects of the fault and are complementary, as they contribute to facilitate the comprehension of the bug in orthogonal way: the first identifies the suspicious pattern of execution that triggers the failure and the second identifies the suspicious input data.

While either technique is obtained by applying well-established debugging approaches (Spectrum-based fault localization [18] and Delta-Debugging [19], [20], respectively), the novelty/originality of MRDebug is in the tailoring of such general-purpose debugging approaches to the specific-domain of *MapReduce* design faults. Spectrum-based fault localization techniques usually locate faulty lines of code, but the *MapReduce* design faults are triggered in non-deterministic way depending on how the framework executes the program. This means that the same faulty code with the same data could sometimes fail, and other times succeed. Therefore, instead of locating the lines of code, the proposed fault localization technique (MRDebug-FL) analyses several executions and characterizes the pattern of the non-deterministic executions that cause the failure: this pattern searched by MRDebug-FL relies on the peculiar characteristics of *MapReduce* configurations.

Besides, in the faulty execution configuration, only a small subset of the input data could be sufficient to trigger the failure and the proposed input reduction technique (MRDebug-IR) iteratively shrinks the test input data until finding a minimal (or reduced) failing subset. MRDebug-IR is obtained by adapting the general-purpose Delta-Debugging technique to reducing the *MapReduce* input data.

The two techniques can be fruitfully integrated so to improve the fault localization efficiency through the execution of fault localization after the input reduction (MRDebug-IR-FL). In this way, the fault localization technique is executed with only the data relevant to understand the fault, because the irrelevant data that can cause noise are first removed by the input reduction technique.

Note that, differently from existing approaches, neither MRDebug-FL nor MRDebug-IR need an oracle, as they both embed our previous test technique called MRTest [14], which can determine whether a test execution passes or fails by applying metamorphic testing (however MRTest by itself does not support fault localization or input reduction). MRDebug also differs from general-purpose debugging techniques as it only requires one test case that is executed several times by incorporating small changes at each execution: according to the Lewis Counterfactual theory of causality [21], the root cause of the fault is the one of these small changes that turns the execution

from success to failure. In summary, the contributions of this paper include:

- MRDebug-FL: A *MapReduce* specific Spectrum-based fault localization [18] technique to obtain the root cause of the fault discovering the characteristic(s) that are common among the executions that trigger the failure.
- MRDebug-IR: A *MapReduce* specific Input reduction technique based on Delta Debugging algorithm [19], [20] to isolate the data that trigger the failure.
- MRDebug-IR-FL: The integration of the Input reduction technique followed by the Fault localization technique aimed to obtain better results than the two techniques applied separately.
- Automation of both fault localization and input reduction techniques without using a user-defined oracle.
- Experimentation with test cases executed against real-world and seeded programs to analyse both the effectiveness and efficiency of the debugging techniques proposed.

The scenario of use of MRDebug is when developer/tester executes locally the unit test cases and one of them triggers a failure caused by a design fault. In that case, MRDebug automatically provides both the root cause of the design fault and the minimal data that trigger the failure.

It is worth noting that each *MapReduce* framework is slightly different because it adopts the *MapReduce* processing model in different ways. The proposed debugging framework, MRDebug, is implemented for *Hadoop MapReduce* applications executed in development environment with smaller data than the production data.

The remainder of this paper is organized as follows. Section II describes the background of *MapReduce*, design faults and fault localization. The debugging framework is presented in Section III. The fault localization technique is defined in Section IV, the input reduction technique in Section V, and the integration of both techniques in Section VI. The experimentation is performed in Section VII. Finally, the related work is discussed in Section VIII and the conclusions are drawn in Section IX.

II. BACKGROUND

To understand the underlying concepts and terminology of the paper, this section introduces the *MapReduce* processing model, the testing technique used by the proposed debugging approach (MRTest [14]), and the bases of fault localization.

A. *MapReduce* Processing Model

The *MapReduce* processing model [6] allows developers to process massive data in parallel through the “divide and conquer” principle. In one of the most basic designs, the developer only needs to implement two *MapReduce* functions: *Mapper* and *Reducer*. The *Mapper* function receives the input and produces $\langle key, value \rangle$ pairs where the *key* is the identifier of one subproblem and the *value* is the part of the data needed to solve it. Next, the *Reducer* receives all values grouped by *key* ($\langle key, [values] \rangle$ pairs) and solves the subproblems. In order to scale to massive data, the program is executed in a *Big Data* cluster on frameworks such as *Hadoop* [2], *Spark* [3] or *Flink* [4],

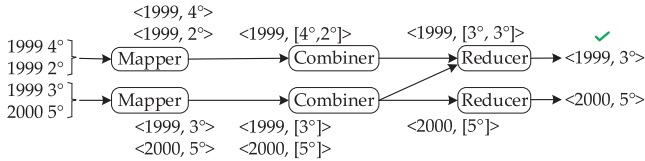


Fig. 1. Execution of MapReduce program.

among others. The program uses the distributed resources available in the cluster to execute several *Mapper* processes in parallel and, when they finish, it also executes several *Reducer* processes in parallel. These parallel executions decrease the execution time, but the high quantities of $\langle \text{key}, \text{value} \rangle$ pairs emitted from *Mapper* to *Reducer* may cause net bottlenecks. This issue can be avoided if the developer designs the program with another *MapReduce* function called *Combiner* (e.g., *combineByKey* in *Spark* or *CombineFunction* in *Flink*). In this scenario, a *Combiner* is executed after each *Mapper* and decreases the number of $\langle \text{key}, \text{value} \rangle$ pairs by solving the subproblems with the data available locally in the *Mapper* process. Once the *Combiner* obtains the partial solutions for each *Mapper*, these partial solutions are emitted to the *Reducers* that obtain the global solutions.

For example, suppose a program that calculates the average temperature per year given an input of “year; temperature” rows. This program can be divided in one subproblem per year (key) and can be designed in the following way: the *Mapper* function emits $\langle \text{year}, \text{temperature} \rangle$ pairs, and both *Combiner* and *Reducer* receive $\langle \text{year}, [\text{temperatures}] \rangle$ and emit $\langle \text{year}, \text{average of temperatures} \rangle$. Depending on the resources available during runtime, the program is executed with different numbers of these *Mappers*, *Combiners* and *Reducers*. Fig. 1 represents a simple execution with 2 *Mappers*, one *Combiner* per *Mapper*, and 2 *Reducers*. First, each *Mapper* receives a subset of years and the temperatures as input, and produces an outcome with one $\langle \text{year}, \text{temperature} \rangle$ pair per input. After each *Mapper* has finished, a *Combiner* receives an input with the $\langle \text{year}, \text{temperature} \rangle$ pairs from the *Mapper*, but grouped by the key ($\langle \text{year}, [\text{subset of temperatures}] \rangle$ pairs). Next, each *Combiner* calculates the average with the partial temperatures. Once all *Mappers* and *Combiners* have finished, the *Reducer* receives inputs composed by one year with all partial averages ($\langle \text{year}, [\text{partial averages}] \rangle$), and finally the *Reducer* calculates the global average per year.

In a development environment, the program is usually executed in one computer without parallelism. However, the execution in the production environment is carried out in several servers and the infrastructure failures are frequent. The execution of the program is automatically managed by a framework that divides the dataset to be analysed in parallel, allocates resources, and handles the common infrastructure failures, among others. Delegating the management of the program execution to a framework is an advantage for developers because they can focus on designing the program functionality without having to deal with the underlying distributed execution. However, this is also a disadvantage for testers because the program should succeed in all possible configurations/executions and

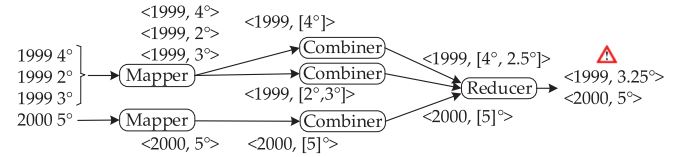


Fig. 2. Failure execution of MapReduce program.

testers do not have fine-grained control of the processes executed because the parallel execution is automatically handled by the framework. Even a faulty *MapReduce* program can sometimes succeed, and other times fail, depending only on how the framework decides to execute the program processes.

For example, Fig. 1 and Fig. 2 represent two different executions of the same program with the same input data, but one of them triggers a failure. The failure is triggered when the execution of *Combiner/Reducer* is not commutative/associative as in the execution of Fig. 2, and succeeds in other cases. The program is wrongly designed because if *Combiner* emits the local average temperatures, then the *Reducer* is not always able to calculate the global average temperature from these local average temperatures: for the year 1999, $\text{avg}(\text{avg}(4^\circ), \text{avg}(2^\circ, 3^\circ)) \neq \text{avg}(4^\circ, 2^\circ, 3^\circ)$ because $3.25^\circ \neq 3^\circ$. The first *Combiner* of Fig. 2 receives 4° and emits 4° ($\text{avg}(4^\circ) = 4^\circ$). At the same time, the second *Combiner* of Fig. 2 transforms the temperatures 2° and 3° into 2.5° ($\text{avg}(2^\circ, 3^\circ) = 2.5^\circ$). Therefore, the *Reducer* receives for the year 1999 the local average temperatures 4° and 2.5° , and emits their average 3.25° ($\text{avg}(4^\circ, 2.5^\circ) = 3.25^\circ$). This *Reducer* (Fig. 2) is incorrect because the average temperature should be 3° ($\text{avg}(4^\circ, 2^\circ, 3^\circ) = 3^\circ$) as happens in Fig. 1.

This kind of faults are referred to as design faults in this paper. A design fault is when the program is not correctly designed in accordance with the *MapReduce* semantics/correctness properties [22] (partition-isolation and commutativity, among others). When a *MapReduce* program is implemented with a design fault, it could execute correctly one time and fail in the next execution even with the same code and data. The program fails intermittently because when the *MapReduce* semantics/correctness properties are not satisfied, the program does not support all possible non-deterministic executions that can happen in production. To avoid design faults, the developer must design the program with the proper *Mapper*, *Reducer* and/or *Combiner* functions in accordance with the *MapReduce* semantics/correctness properties. There are different types of design faults, some of them are introduced when the developer tries to optimize the program with a wrong *Combiner*, for example implementing a *Combiner* functionality that is not idempotent, or *Combiner-Reducer* functions that are not commutative/associative. Other design faults can be introduced by the wrong assumptions about the data placement, for example when the *Combiner/Reducer* functions expect the data sorted in some way, but they are not due to the distributed execution, or when one of the *Mapper/Combiner/Reducer* functions requires some data that are in another computer.

Design faults are not only difficult to detect, but also to debug. Some faults could be fixed changing few lines of code, but other faults can require to change much more code or even add/remove some *MapReduce* functions. For example, if the

fault is in *Combiner*, the developer can easily fix it by removing the *Combiner* function: in *Hadoop Streaming* this only needs to remove the parameter call “-combiner”, in *Hadoop MapReduce* to remove the API call “setCombinerClass”, or in *Flink* to remove the “combine” function from a *Reducer* class called *reduceGroup*. However, the previous changes are not the best solution because the fixed program will not execute the *Combiner* optimizations. Another way to fix the fault is redesigning the *Mapper-Combiner-Reducer* functions: in *Flink* program the developer can implement the *Reducer* interface “GroupCombineFunction” by including/changing the *Combiner* code, in *Spark* by changing the *Reducer* function “reduceByKey” to another *Reducer* function called “aggregationByKey” with the new *Combiner* code. In the faulty program that calculates the average temperature per year, the fault can be fixed removing the *Combiner*, or redesigning the whole program. In this last case, the developer can change the *Combiner* so that instead of calculating the partial average of temperatures, it calculates both the partial sum of temperatures and also the number of temperatures added together. Despite this change seems simple, it requires to re-design the whole program: *Mapper* now should emit $\langle year, \{temperature, 1\} \rangle$ to follow the same syntax and semantic of *Combiner*, and *Reducer* now should implement a new logic because it does not receive temperatures but both sums and counts of temperatures.

We are not aware of any research survey that analyses how prevalent are the design faults in the *MapReduce* programs, but an analysis of 507 programs reveals that at least 5 programs have a design fault and 58% of *Reducers* do not satisfy the commutative property [11]. The programs of these *Reducers* could also have a design fault depending on the *Combiner* code if any. There are also other design faults beyond the non-commutativity of *Combiner/Reducer* [12], [13].

B. Automatic Testing in MapReduce: MRTTest

In previous work [14], we devised a testing technique called MRTTest that is able to automatically detect the design faults of the *MapReduce* applications using Partition [23], Combinatorial [24], [25] and Metamorphic Testing [26], [27]. MRTTest receives test input data and detects a fault when different executions of the same program on the same data do not yield the same outputs. It is not feasible to execute all possible executions that can occur in production, but MRTTest executes a representative subset of them. After the execution, MRTTest checks automatically that the outputs of all executions are equal. MRTTest and their underlying concepts (design fault, characteristics, configurations and execution) are used in the debugging approach proposed in this paper to both enhance and automate the debugging information about the design faults.

Design fault: A fault of the *MapReduce* programs whose functionality can be rightly implemented but does not satisfy the semantics/correctness of the *MapReduce* programming model. A wrongly designed program can obtain two different outputs if the program is executed twice. This is because the program is not correctly designed to support all possible executions that could happen in production. Given a set S of all semantics that all *MapReduce* programs must satisfy (e.g. partition-isolation,

among others [22]), a *MapReduce* program P has a design fault when the program does not satisfy some of these semantics: $\exists s \in S \mid P \not\models s$. A design fault could be triggered non-deterministically depending on how the framework manages the distributed execution of the program.

Characteristic: A characteristic ch_i defines an execution pattern of a *MapReduce* program in a parallel framework, for example, “>1 Mapper”. The set of characteristics of a *MapReduce* program (CH) is obtained by means of partitioning [23], considering the execution structure and the data distribution of the *MapReduce* program as the input domain, where each partition is a characteristic. In total, for the *Hadoop MapReduce* framework CH has the following 17 characteristics¹: “1 Mapper”, “>1 Mapper”, “data executed in the same order as in the input”, “data executed in different order as in the input”, “data equally distributed in the Mappers”, “data non-equally distributed in the Mappers”, “0 Combiner”, “1 Combiner”, “>1 Combiner”, “Mapper output equally distributed in the Combiners”, “Mapper output non-equally distributed in the Combiners”, “0 data directly from Mapper to Reducer”, “>0 data directly from Mapper to Reducer”, “1 iterative executions of Combiner”, “>1 iterative executions of Combiner”, “1 Reducer” and “>1 Reducer”. More details about these characteristics are given in [14].

Configuration: A configuration $conf_i = \langle ch_1, ch_2, \dots \rangle$ is a combination of characteristics that represents a potential execution of a *MapReduce* program in a parallel framework. For example, the configurations $conf_1 = \langle > 1 \text{ Mapper}, \dots, 1 \text{ Reducer} \rangle$ and $conf_2 = \langle > 1 \text{ Mapper}, \dots, > 1 \text{ Reducer} \rangle$ represent the *MapReduce* program execution of Figs. 1 and 2, respectively. The set of configurations used in MRTTest to test the *MapReduce* program is denoted $CONF$ and it is generated applying combinatorial testing [24], [25] to the set of the characteristics. In 2-wise combination there are 11 different configurations. Note that not all characteristics can be combined in the same configuration due to possible constraints between them. For example, a configuration with “1 Combiner” must have “Mapper output equally distributed in the Combiners”, because there is only 1 *Combiner*. The configurations generated with the combinations and their constraints are both defined and detailed in [14]. A special case of configuration is the base configuration $conf_{base}$, that represents an execution of the *MapReduce* program with neither parallelism nor optimizations.

Execution: Given a test input data t and a configuration $conf_i$, an execution $exec(t, conf_i)$ is the distributed flow of *Mapper*, *Combiner* and *Reducer* processes executed according to the configuration $conf_i$ and t as input. The execution $exec(t, conf_i)$ produces an output formed by the $\langle key, value \rangle$ pairs emitted by this program execution. For example, Figs. 1 and 2 show the $exec(t, conf_1)$ and $exec(t, conf_2)$, respectively, which produce different outputs.

MRTTest aims to check if a program has a design fault or not. To this end, MRTTest receives a test input data, generates the configurations, executes the program covering these configurations, and finally checks that all of these executions produce the

¹Other *MapReduce* frameworks (e.g. *Spark* or *Flink*) could have different characteristics.

```

function MRTest (t)
Input : A test input data t
Output: PASS/FAIL. When FAIL the output is also the configuration
BaseOutput ← exec(t, confbase)
CONF ← GenerateConf(CH) # e.g. 2 – Wise
for each confi ∈ CONF
  { ExecutedOutput ← exec(t, confi) # follow – up execution
  do { if ExecutedOutput ≠ BaseOutput
      then output (FAIL, confi)
  }
output (PASS)

```

Fig. 3. MRTest pseudo-code.

same output. The MRTest technique is summarized in the pseudo-code of Fig. 3.

First, MRTest executes the program according to the most basic configuration ($conf_{base}$) and saves the output (base output). Next, MRTest generates other configurations and executes the program according to these configurations. There are several potential executions per each configuration because the same characteristic can be covered in different ways, for example “>1 Mapper” can be covered in executions with 2 Mappers or 3 Mappers, among others. For each configuration generated, MRTest executes randomly the program yet assuring that the whole configuration is covered. Each one of the executions is called a follow-up execution according to the terminology of Metamorphic testing [27]. Finally, MRTest compares the output of each execution against the base output (metamorphic relation). In case one of the executions produces different output, then MRTest reports a failure due to a design fault because the fault is triggered or masked depending on how the program is executed. In the reporting of failure, MRTest also indicates the configuration that triggers the failure. Note that MRTest is able to detect the design faults automatically for every *MapReduce* program without knowing either the expected output or the semantics of the program. According to our previous experiments [14], MRTest is accurate and can be used as automatic partial oracle [28].

For example, suppose that a tester wants to test the *MapReduce* program that calculates the average temperature per year with the *Mapper*, *Combiner* and *Reducer* functions described in Section II-A (Fig. 1 and Fig. 2). The tester designs a test case with 4 input data composed by year 1999 with 4°, 2° and 3°, and year 2000 with 5°. MRTest automatically generates the 11 configurations based on the 2-wise combination of the characteristics. Next, MRTest executes the test case following these configurations (follow-up executions), and checks if all of them produce the same output. Two of the executions are depicted in Fig. 1 and Fig. 2 and their outputs are different, so MRTest has automatically detected a design fault.

C. Debugging Techniques

There are several debugging techniques that help the tester in different ways, among others the fault localization and the input reduction techniques. While fault localization helps testers locate the root cause of the fault, the input reduction techniques help them understand the fault by providing the minimal input

data that trigger the failure. The main concepts of these two techniques are detailed below.

The **fault localization** technique most used in research is the spectrum-based fault localization that analyses the common behaviour between the test executions that fail and the different behaviour in those that succeed [18]. For example, if all test cases that fail cover one line that is not covered in successful test cases, then this line is suspected as the cause of the failures. The spectrum-based fault localization techniques obtain a ranking of the most suspicious causes of the fault according to the following procedure: (1) definition of the behaviour to be analysed, e.g., line coverage (program spectra), (2) generation of several test cases, (3) execution and monitoring of the test cases, and (4) analysis of the behaviours observed during the execution.

The observed behaviour that is used to find the root cause of the fault is defined according to program spectrum [29], [30] such as the coverage of code, parts of the execution, or other execution-related data. For example, if the program spectra used are the lines of code, then spectrum-based fault localization observes the lines covered and not covered by the test cases and obtains a ranking of the lines that are most suspicious to trigger the failures. The first line of the ranking is the most suspicious to be the root cause of the fault. This ranking is called “suspiciousness ranking” and is obtained analysing the similarity/distance from the vector that contains the failures of all test cases (failure) to the vector of the coverage of each line in all executions (behaviour covered). Those lines that are both covered during the failures and not covered in the succeeded test cases, are more likely to be the root cause of the fault. That is, the line that has the highest suspiciousness to be the root cause of the fault is the one that has the most similar vector of coverage to the failure vector. There are several ranking metrics in the literature that can be used to calculate this similarity, but no single one is the best in all domains of fault localization [31].

Many fault localization techniques proposed in the literature obtain the suspiciousness ranking analysing the source code, like the lines or branches covered by the test cases. However, the root cause of the fault is not always the source code. For example, the localization techniques of product lines locate the root cause of the faults in features sets instead of the source code [32]. In the case of the design fault of *MapReduce* programs, the root cause of the fault is not the code itself either because the same faulty code sometimes triggers the failure and other times masks it depending only on how the external framework executes this code. Locating which are the patterns in common among those executions that trigger the failure could help the tester to understand the design fault and fix it. Therefore, the root cause of the fault is the characteristic that have in common the configurations that expose the design fault, such as the number of *Mappers* executed in parallel or the number of *Combiners*. Once the fault has been located, the developer would fix the program redesigning it usually by changing several faulty lines of code and adding/removing new operations.

Input reduction techniques use different strategies to reduce the data until a minimal data that triggers the failure is obtained. One of these strategies is Delta Debugging [19], [20] that finds the first local minimum using recursively a greedy algorithm

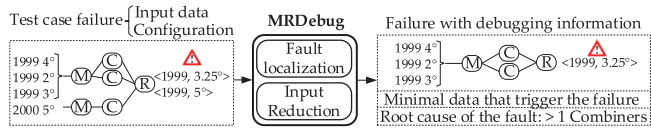


Fig. 4. MapReduce debugging framework.

with a binary-search. In the first step, Delta Debugging divides the test input data in two halves (binary-search). If one of the halves still triggers the failure, then this half is again divided further with the binary-search approach and the other half is discarded (greedy). However, if neither of the two new halves triggers the failure (i.e., the failure is triggered by a mix of the two halves), then the granularity of the search space is increased: one of the two halves is selected together with one half of the other half. This approach is applied recursively until the first local minimum is reached.

III. DEBUGGING FRAMEWORK: MRDEBUG

This paper describes the MRDebug framework that aims to help testers during the debugging of the *MapReduce* design faults. This framework, depicted in Fig. 4, automatically locates the root cause of the design faults and provides the minimal data that trigger the failure. To this extent, MRDebug combines two orthogonal debugging techniques: fault localization (MRDebug-FL) and input reduction (MRDebug-IR). These debugging techniques start with a test case failure: test input data and configuration that trigger the failure (e.g., obtained by MRTest during the testing). Both techniques execute MRTest several times as automatic partial oracle to obtain valuable information to debug the programs automatically. The fault localization and the input reduction techniques can be executed separately or in combination (MRDebug-IR-FL):

- MRDebug-FL: reveals the characteristics that cause the design fault (root cause of the fault). For example, if MRDebug-FL indicates “>1 *Combiner*” as root cause of the fault for the program that calculates the average temperature per year, this means that the test case usually fails in configurations with “>1 *Combiner*” and also does not usually fail in other configurations. Despite the fact that the characteristic “>1 *Combiner*” is the root cause of the fault, this fault is not always triggered with “>1 *Combiner*”, as the failures depend not only on the configuration executed but also on the input data executed.
- MRDebug-IR: obtains the minimal data of the test case that still trigger the failure. For example, in Fig. 4 the test case reveals the fault with 4 <key, value> pairs, but MRDebug-IR indicates that the failure can still be triggered with only 3 <key, value> pairs. In this example MRDebug-IR has only reduced the input data of 1 <key, value> pair (from 4 to 3), but note that the example is illustrative and MRDebug-IR is able to reduce several <key, value> pairs until reaching minimal data that still triggers the failure.
- MRDebug-IR-FL: first reduces the data (MRDebug-IR) and then localizes the root cause of the fault (MRDebug-

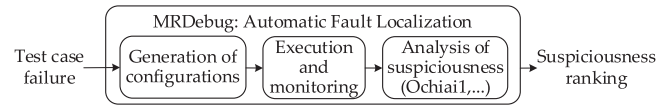


Fig. 5. Fault localization technique in MapReduce programs.

FL) among these reduced data. Executing both techniques together is more effective than executing them separately because the fault localization in MRDebug-IR-FL only receives the data that cause the failure and does not receive noise from other irrelevant data.

Using multiple debugging techniques could benefit the developers [17]. In the case of MRDebug, the two techniques of fault localization and input reduction provide complementary information because they are focused on debugging different parts. The fault localization technique is focused on the characteristic of the execution that triggers the failure, and the input reduction technique is focused on the minimal data that trigger the failure. This combined debugging information obtained automatically by MRDebug can help testers understand the fault.

MRDebug debugs *MapReduce* applications implemented for *Hadoop MapReduce* framework. During the debugging, the unit test cases are re-executed several times in a local environment to guarantee both a fine-grain control and reproducibility. In production environment it would be quite difficult to guarantee the previous because the framework manages the execution in a non-deterministic way according to the resources available. *Hadoop MapReduce* has more than 190 parameters that can affect the execution [33] and the official documentation indicates that some of these parameters interact subtly with the execution making the fine-control even more complex [34].

The deployment scenario of MRDebug is when a unit test case fails due a design fault. The tester uses MRDebug to obtain automatically information related to the fault: either of the two provided debugging techniques can be used alone, or they could be used in combination (in which case we suggest to first apply data reduction and then fault localization). The tester uses the fault localization technique, MRDebug-FL, to obtain the information related to the non-deterministic executions that trigger the failure. On the other hand, the tester uses the input reduction technique, MRDebug-IR, to obtain information related to which part of the input data is the one that triggers the failure. The tester could also use both techniques together with MRDebug-IR-FL to obtain information of the fault related to both data and execution. Section IV details further the fault localization technique, Section V the input reduction technique, and Section VI the combination of both techniques.

IV. MRDEBUG-FL: FAULT LOCALIZATION

The spectrum-based fault localization proposed in this paper, MRDebug-FL, uses the characteristics as program spectra with the goal of automatically identifying which characteristic causes the design fault in the *MapReducer* programs. As Fig. 5 summarizes, MRDebug-FL receives a test case failure, and provides a ranking of the most suspicious characteristics to cause the failure. To this end, MRDebug-FL starts generating several


```

function MRDebug-FL ( $t, conf_{fail}$ )
Input : A test input data  $t$ , and a configuration that causes failure  $conf_{fail}$ 
Output: Ranking of characteristics
# Generation of configurations
 $CONF_{FL} \leftarrow \{conf_{fail}\}$ 
for each  $ch_i \in conf_{fail}$ 
  do  $\begin{cases} conf_{FL} \leftarrow \text{ChangeCharacteristic}(conf_{fail}, ch_i) \\ \text{ADD}(conf_{FL}, CONF_{FL}) \end{cases}$ 
# Execution and monitoring
 $EXEC\_FAILURES \leftarrow \{\}$ 
 $BaseOutput \leftarrow \text{exec}(t, conf_{base})$ 
repeat until K times # parameter defined by the tester
  for each  $conf_{FL} \in CONF_{FL}$  # MRTest
    do  $\begin{cases} ExecutedOutput \leftarrow \text{exec}(t, conf_{FL}) \\ \text{if } ExecutedOutput \neq BaseOutput \\ \text{then } \text{ADD}(conf_{FL}, EXEC\_FAILURES) \end{cases}$ 
# Analysis of suspiciousness
 $RANKING \leftarrow \{\}$ 
for each  $ch_i \in CH$ 
  do  $\begin{cases} COVERAGE\_CH_i = \text{GetCoverage}(ch_i, CONF_{FL}) \\ suspiciousness_{ch_i} \leftarrow \text{RankingMetric} \left( \begin{matrix} COVERAGE_{CH_i} \\ EXEC\_FAILURES \end{matrix} \right) \\ \quad \quad \quad \# \text{ ranking metric selected by the tester} \\ \text{ADD}(ch_i, RANKING) \text{ sorted by } suspiciousness_{ch_i} \end{cases}$ 
output ( $RANKING$ )

```

Fig. 6. MRDebug-FL pseudo-code.

configurations varying their characteristics (Generation of configurations). Next, the test input data is executed according to these configurations using MRTest in order to monitor if the execution succeeds or not (Execution and monitoring of configurations). Finally, MRDebug-FL analyses with a ranking metric which characteristics are more frequently executed in the configurations that produce a failure and which ones in those that succeed (Analysis of suspiciousness). MRDebug-FL executes the algorithm of Fig. 6 as described below.

Generation of configurations: The configurations are generated based on the Lewis counterfactual theory of causality [21] like other fault localization techniques [35]. In such a way, MRDebug-FL generates new configurations ($CONF_{FL}$) where each new configuration ($conf_{FL}$) has the same characteristics as the configuration that failed in testing ($conf_{fail}$), bar one that is changed. For example, if $conf_{fail} = \langle \text{">1 Mapper"}, ch_2, \dots \rangle$, then a new configuration $conf_{FL} = \langle \text{"1 Mapper"}, ch_2, \dots \rangle$ is generated changing only ch_1 , and other new configurations are generated in the same way, changing each other ch_i . The goal of this single change is to check whether the characteristic changed between the $conf_{fail}$ and the $conf_{FL}$ is able to mask the fault or continue to fail. In both cases, the configuration $conf_{FL}$ provides clues about the root cause of fault.

Fig. 7 summarizes the generation of these new configurations for the program of Section II-A. First, the configuration $conf_{fail}$ fails during testing with the following characteristics: $conf_{fail} = \langle \text{">1 Mapper"}, \dots, \text{">1 Combiner"}, \dots \rangle$. MRDebug-FL generates new configurations ($CONF_{FL}$) varying each characteristic: the " >1 Mappers" is changed to " 1 Mapper" in $conf_{FL} A$, and " >1 Combiner" is changed to

" 0 Combiners" in $conf_{FL} B$. Note that each new configuration $conf_{FL}$ only changes one characteristic in comparison with $conf_{fail}$ and the other characteristics of $conf_{FL}$ remain the same. The figure only shows two $conf_{FL}$ (A and B), but there is one $conf_{FL}$ per each characteristic changed.

Execution and monitoring: For each configuration $conf_{FL}$ generated in $CONF_{FL}$, MRDebug-FL executes the program with the input data of the test case t , and monitors if the execution triggers a failure or not. This execution is carried out by MRTest because it is able to both detect the design faults automatically and guarantee that the program is executed according to all the characteristics of $conf_{FL}$. In the best case, the characteristic that causes the fault would trigger the failure each time that a configuration with this characteristic is executed, but this is not always true in practice due to the coincidental masking of the faults. To avoid this problem, MRDebug-FL executes K times each $conf_{FL}$, where K is a tester supplied parameter (default is $K = 5$). Note that a configuration can be executed in different ways, for example a configuration $conf_{FL} = \langle \text{">1 Mapper"}, ch_2, \dots \rangle$ must be executed with more than one *Mapper*, and this is possible in different ways: 2 *Mappers*, 3 *Mappers*, or 2 *Mappers* but different data in each one, among others. Each configuration $conf_{FL}$ is executed K times randomly but forced to cover the characteristics of the $conf_{FL}$. The more the executions, the better the results, but at the expense of more execution time.

In the example depicted in Fig. 7, one execution triggers a failure (the execution of $conf_{FL} A$), but another execution masks the fault (the execution of $conf_{FL} B$). Note that Fig. 7 only illustrates one execution of two $conf_{FL}$.

Analysis of suspiciousness: After finishing all executions, MRDebug-FL obtains the suspiciousness ranking by analysing the following information of these executions: if each execution succeeds or fails, and the characteristics executed in each execution. This analysis is done by a ranking metric that obtains per each characteristic how much it is suspicious, and then ranks these characteristics from more to less suspicious. The ranking metrics usually consider that one characteristic is more suspicious of causing the fault when the executions that execute this characteristic fail. In the same way, a characteristic is also suspicious when the executions that succeed do not execute this characteristic. The ranking metrics compute the suspiciousness of each characteristic ($suspiciousness_{ch_i}$) based on the similarity of two binary vectors: one of the vectors indicates per each execution if the failure was triggered or not ($EXEC_FAILURES$), and the other vector indicates also per each execution if the characteristic was executed or not ($COVERAGE_CH_i$). The more the vectors are similar, the more the characteristic is suspicious, because either the executions usually fail when the characteristic is executed, or the executions usually do not fail when the characteristic is not executed. MRDebug-FL implements the most common ranking metrics from the literature [18], [36], [37] (there are 52 of them), and each one computes the $suspiciousness_{ch_i}$ based on the existing notions of similarity between two binary vectors, but they weight the computation in different ways. One of these ranking metrics is Ochiai1 and its computations is $suspiciousness_{ch_i} = N_{CF} / \sqrt{N_F \cdot (N_{CF} + N_{CS})}$, where N_F is the number of failing executions, N_{CF} is the number

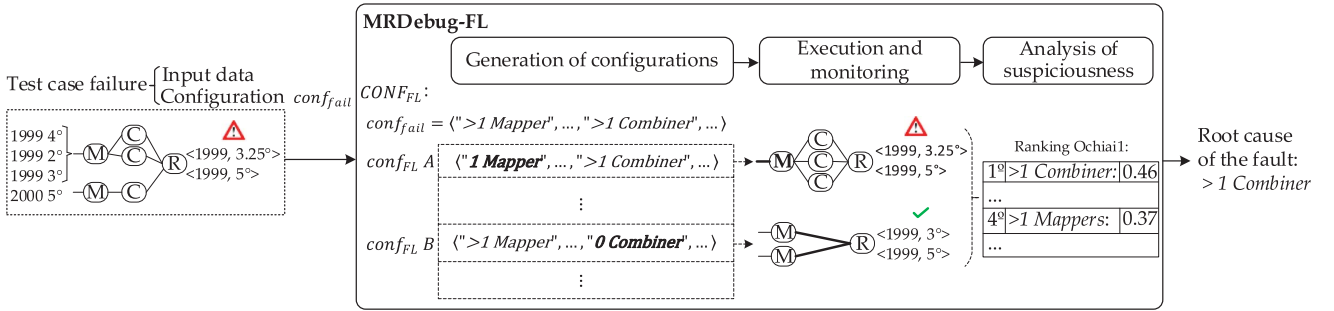


Fig. 7. Example of the fault localization technique.

of failing executions that cover ch_i , and N_{CS} is the number of successful executions that cover ch_i . Other ranking metrics also compute N_S (successful executions), N_C (executions that cover ch_i), N_U (executions that do not cover ch_i), N_{US} (successful executions that do not cover ch_i), and N_{UF} (failing executions that do not cover ch_i). Finally, the first positions of the suspiciousness ranking contain those characteristics that are most suspicious to cause the design fault according to one of the 52 ranking metrics selected by the tester.

In the example represented in Fig. 7, MRDebug-FL obtains the suspiciousness ranking based on the Ochiai1 metric, comparing the similarity between the EXEC_FAILURES vector and the coverage vector of each characteristic: “1 Mapper”, “>1 Mapper”, and the others not depicted in Fig. 7. The vector of “>1 Mapper” is different than the vector of failures because despite some executions with “>1 Mapper” fails (the execution of $conf_{fail}$), there are other executions that succeed with “>1 Mapper” (the execution of $conf_{FL B}$). According to the Ochiai1 ranking metric, the suspiciousness of “>1 Mapper” is 0.37. We cannot say if this suspiciousness is high or low because it depends on how much more/less suspicious other characteristics are. In the case of “>1 Combiner”, the coverage vector is quite similar to the vector of failures because the executions that fail usually execute “> 1 Combiner” (executions of $conf_{fail}$ and $conf_{FL B}$). The suspiciousness of “>1 Combiner” according to Ochiai1 is 0.46. Finally, MRDebug-FL ranks the characteristics based on the computed suspiciousness, and “>1 Combiner” is the characteristic most suspicious to cause the design fault. Note that the suspiciousness of “>1 Combiner” is not 1 because there are some executions not depicted in Fig. 7 that succeed when executing “>1 Combiner” due to coincidental masking.

V. MRDEBUG-IR: INPUT REDUCTION

Given a test case failure, the input reduction technique, MRDebug-IR, applies Delta Debugging to recursively select only a subset of the input data until a minimal input data that still triggers the failure is reached. The reduction technique executes the algorithm of Fig. 8 as described below. This algorithm is similar to Delta Debugging but with slight differences. During the reduction, MRDebug-IR uses MRTTest to check whether the reduced data still trigger the failure, and also allows to stop the reduction when the test input data is small enough. To this end,

MRDebug-IR has an optional parameter “threshold” in case the tester wants to indicate how much test input data is small enough, regardless of whether it is possible to reduce the data further.

In the first part of the reduction, the test input data are divided into two subsets with half of the data each (binary-search). Next, MRDebug-IR executes each subset using MRTTest [14] as automated partial oracle in order to check automatically if the subset of the test input data still triggers the failure or not. If one of these subsets triggers the failure, then this subset is divided again. In the case that neither of the two subsets of data triggers the failure, the granularity is increased (e.g., one of the subsets together with half of the other subset). The algorithmic details of how MRDebug-IR works are described below.

MRDebug-IR executes recursively the input reduction algorithm trying in each iteration to reduce a little bit the subset of the test input data that triggers the failure. This subset is represented in the algorithm by $t_{reduced}$ and in the first iteration it is the whole test input data. MRdebug splits this subset into numberKVplits sub-subsets of it called splits. In the first iteration, numberKVplits is 2, which means that the test input data is divided into two splits. Each time one of the splits triggers a failure, the algorithm tries to reduce recursively this faulty split into 2 splits. When none of these 2 splits triggers a failure, the faulty data is partially assigned in one split and partially in the other split. In that case, the algorithm increases the granularity analyzing again the same subset of data but with the double of numberKVplits. Next, MRDebug analyzes each one of the splits of the subset (KVsubset) together with their complements (KVcomplement). A complement is the subset minus each one of the splits, this means that the first time the complement is one half of the subset together with one half of the other half of the subset. The algorithm is executed in this way recursively until either it reaches the minimal data or exceeds the threshold. A minimal test input data is reached when the subset that triggers the failure cannot be split in more parts because the number of splits is already bigger or equal than the number of $\langle key, value \rangle$ pairs of the subset.

Fig. 9 illustrates how MRDebug-IR reduces the input data from a test case that fails. First, the 4 $\langle key, value \rangle$ pairs of the input data are divided into two halves: the first half has the first 2 pairs ($t_{reduced A}$), and the second half the other 2 pairs ($t_{reduced B}$). Next, each half is executed using MRTTest, but


```

function MRDebug-IR ( $t, conf_{fail}$ )
Input : A test input data  $t$  that triggers failure, a configuration that causes
failure  $conf_{fail}$ , and optionally a threshold indicating how much
data is minimal enough to stop the reduction
Output: Minimal subset of  $t$  (or at least lower than the threshold) that still
triggers the failure, and a configuration that triggers the failure with
the reduced data

 $t_{reduced} \leftarrow t$ 
 $conf_{fail\ reduced\ data} \leftarrow conf_{fail}$ 
 $numberKV\ splits \leftarrow 2$ 
output (InputReduction( $t_{reduced}, numberKV\ splits$ ))

function InputReduction ( $t_{reduced}, numberKV\ splits$ ) # recursive function
Input : A subset of the test input data  $t_{reduced}$ , and a  $numberKV\ splits$ 
that indicates the number of splits in which the test input data
 $t_{reduced}$  should be split
Output: Minimal subset of  $t$  (or at least lower than the threshold) that still
triggers the failure, and a configuration that triggers the failure with
the reduced data

if  $numberKeyValuePairs(t_{reduced}) \leq threshold$ 
  then output ( $t_{reduced}, conf_{fail\ reduced\ data}$ )
 $keyValueSubsets \leftarrow SplitKeyValuePairs(t_{reduced}, numberKV\ splits)$ 
for each  $KVsubset \in keyValueSubsets$ 
  do  $\left\{ \begin{array}{l} veredict, conf_{fail} \leftarrow MRTEST(KVsubset) \\ \text{if } veredict = FAIL \\ \text{then } \left\{ \begin{array}{l} conf_{fail\ reduced\ data} \leftarrow conf_{fail} \\ \text{output} (InputReduction(t_{reduced}, 2)) \end{array} \right. \end{array} \right.$  # reduce to subset

for each  $KVsubset \in keyValueSubsets$ 
  do  $\left\{ \begin{array}{l} KVcomplement \leftarrow t_{reduced} - KVsubset \\ veredict, conf_{fail} \leftarrow MRTEST(KVcomplement) \\ \text{if } veredict = FAIL \\ \text{then } \left\{ \begin{array}{l} conf_{fail\ reduced\ data} \leftarrow conf_{fail} \\ \text{output} (InputReduction(KVcomplement, \right. \\ \left. \max(numberKV\ splits - 1, 2))) \end{array} \right. \end{array} \right.$  # reduce to complement

if  $numberKV\ splits < NumberKeyValuePairs(t_{reduced})$ 
  then output (InputReduction( $t_{reduced}, 2 * numberKV\ splits$ ))
  # increase granularity

  else output ( $t_{reduced}, conf_{fail\ reduced\ data}$ ) # reduction done

function NumberKeyValuePairs ( $t_{reduced}$ )
Input : A test input data  $t_{reduced}$ 
Output: the number of  $\langle key, value \rangle$  pairs that have the  $t_{reduced}$ 
 $numberKV \leftarrow 0$ 
for each  $KVpair \in t_{reduced}$ 
  do  $numberKV \leftarrow numberKV + 1$ 
output ( $numberKV$ )

```

Fig. 8. MRDebug-IR pseudo-code.

neither of them triggers the failure. Therefore, MRDebug-IR increases the granularity because the failure might be triggered with some data of the first half and other data of the second half. MRDebug-IR starts with the first half together with the half of the second half ($t_{reduced} C$), this means a subset of the three first $\langle key, value \rangle$ pairs. This subset is executed with MRTest and reveals the failure with only three $\langle key, value \rangle$ pairs ($t_{reduced} C$). Finally, MRDebug-IR is executed recursively in order to check if it is possible to reduce more $\langle key, value \rangle$

pairs, but in this illustrative example it is not possible because it has reached the minimum test input data.

VI. MRDEBUG-IR-FL: INPUT REDUCTION AND FAULT LOCALIZATION

MRDebug-IR-FL debugs automatically the *MapReduce* programs with a sequence of input reduction followed by fault localization. The goal of MRDebug-IR-FL is to improve the localization of the design faults using input reduction to remove those $\langle key, value \rangle$ pairs that are not relevant to trigger the failure.

First, the input reduction technique, MRDebug-IR, reduces automatically the input data until either a threshold configured by the tester or a minimal data that triggers the failure ($t_{reduced}$) is reached. Next, the fault localization technique, MRDebug-FL, localizes the design fault automatically (RANKING). There are some potential benefits to execute the fault localization with the reduced data. The fault localization could be more effective with reduced data because these data have less noise and therefore produce less coincidental masking. On the other hand, the fault localization with less data could be faster than with the full data. The experiments of Section VII-C confirm that the fault localization of MRDebug-IR-FL is both more effective and more efficient than MRDebug-FL.

In the example used in the previous sections, the test input data have 4 $\langle key, value \rangle$ pairs (Fig. 4). MRDebug-IR-FL executes MRDebug-IR and obtains that the failure can be triggered with only the 3 $\langle key, value \rangle$ pairs represented in Fig. 9 ($t_{reduced} C$). Next, MRDebug-IR-FL executes MRDebug-FL with these 3 $\langle key, value \rangle$ pairs and obtains that the root cause of the fault is “> 1 Combiner”.

VII. EVALUATION

The goal of the experiments is to evaluate both the effectiveness and the efficiency of MRDebug-FL, MRDebug-IR and MRDebug-IR-FL in debugging the *MapReduce* design faults. Regarding MRDebug-FL, we aim to answer the following research questions:

- RQ1.1. Is MRDebug-FL more effective at locating the root cause of *MapReduce* design fault than a random localization (baseline)?
- RQ1.2. In what position of the suspiciousness ranking obtained by MRDebug-FL is the root cause of the *MapReduce* design fault?
- RQ1.3. How much execution time does MRDebug-FL employ to locate the *MapReduce* design fault?
- RQ1.4. Can MRDebug-FL achieve a trade-off between the design faults located and execution time by varying the number of times that each configuration is executed (parameter K)?

The research questions about MRDebug-IR are:

- RQ2.1. Is MRDebug-IR more effective at isolating the test input data that trigger the *MapReduce* design faults than a random reduction (baseline)?

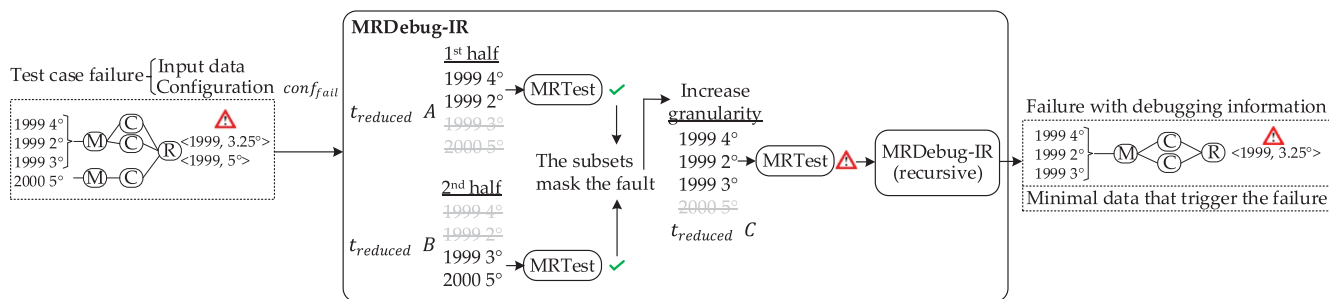


Fig. 9. Example of input reduction.

RQ2.2. How effective is MRDebug-IR in reducing the test input data of the *MapReduce* design faults?

RQ2.3. How much execution time does MRDebug-IR employ to reduce the test input data of the *MapReduce* design faults?

The research questions about MRDebug-IR-FL are:

RQ3.1. Is MRDebug-IR-FL more effective at locating the *MapReduce* design faults than MRDebug-FL?

RQ3.2. Is MRDebug-IR-FL able to decrease the execution time of MRDebug-FL?

RQ3.3. How much data should MRDebug-IR-FL reduce to localize the faults in both a more effective and more efficient way?

To answer the research questions, we have selected the subject programs using Purposive sampling [38] which is the sampling technique most used in recent high-quality software engineering research [39]. In total, we have selected 13 subject programs (P1-13): 5 are real-world programs that have a known design fault (P1-P5), and the other 8 programs (P6-P13) are from *MapReduce* benchmarks used in the literature [40], [41] in which we have injected a design fault. The method followed for the fault seeding is Expert Human Seeder [42], which proposes to inject faults manually in order to cover the different types of design faults based on the knowledge about both *MapReduce* and the type of faults. The faults injected represents a wide variety of design faults according to the different *MapReduce* semantics, grouped in the following fault categories: commutativity/associativity/idempotency (P3, P6 and P12), wrong assumptions regarding to the order of the information in the input/intermediate data (P1, P2 and P13), *Shuffle* phase distributing the data incorrectly in different *Reducers* (P7, P8 and P10), and *Mappers* wrongly partitioned/processed (P4, P5, P9 and P11). Moreover, each one of the faults has been injected to fail in very different way than others from the same category using different *MapReduce* functions/functionalities: *Mapper*, *Combiner*, *Reducer*, *In-Mapper Combiner*, composite key, composite value, setup, or cleanup, among others. In the following, we provide a description of both functionality and faults of subject programs:

(P1) *Open Ankus* [43] is a recommendation system that recommends to the users some items like books based on those preferred by similar users. The *Reducer* incorrectly matches the first item of the first user with the first prediction done even when the prediction is for another item. The program fails

when data about the same both user and item are unsorted due the parallel execution of the framework.

(P2) *Data quality analysis* [44] measures the quality of the data interchanged by companies. The program fails when some particular data under analysis are not processed in the same order as in the input due to parallelization issues done by the framework. The fault is actually fixed.

(P3) *Movie analysis* [45] obtains statistics about movies. The *Combiner* is incorrectly implemented with a non-commutative/non-associative functionality. The program fails when several *Combiners* are executed with the partial data of the same movie and *Reducer* is not able to aggregate them.

(P4) *Data cleaner Knn analysis* [46] cleans the data using a machine learning program to make clusters of similar data. The *Mapper* incorrectly tries to access the non-local available data. The program fails when the framework splits the data of the same cluster in several *Mappers* and the *Mapper* is not able to access all of them.

(P5) *PageRank* [47] calculates the importance/popularity of a web page based on the links connected. *Reducer* requires the information of the web page and all its links, but the *Mapper* incorrectly returns to the *Reducer* only the links that are locally available in this *Mapper*. The program fails when the framework parallelizes the links of a web page in several *Mappers* and *Reducer* only receives part of the links.

(P6) *Wordcount* counts the number of times each word is repeated in a text. The *Combiner* is incorrectly implemented with a non-idempotent functionality that instead of increasing the counter by 1, sets the counter to 1 after the second iterative execution. The program fails when the framework executes the *Combiner* iteratively several times.

(P7) *Grep* counts the times that a pattern is repeated in a text. The *Mapper* emits a composite key containing the grep pattern, but the *Shuffle* phase is incorrect because it does not guarantee that all data of the same pattern go to the same *Reducer*. The program fails when the partial counts of the same grep pattern are in different *Reducers* due the parallel execution of the framework.

(P8) *Flyinghours* finds the total flying hours per departure hour-airport. The *Mapper* incorrectly emits the departure hour as a value instead of as part of composite key. The program fails when a *Reducer* does not receive all data of the same departure hour-airport due the parallelization of the framework.

(P9) *CommuteModeTrips* calculates the number of trips done by each transport mode. The program has an *In-Mapper Combiner* and the *Reducer* incorrectly does not aggregate the partial data from this type of *In-Mapper Combiner*. The program fails when the input data of the same transport mode is split by the framework in several *Mappers* and *Reducer* does not aggregate them.

(P10) *MovieAge* counts the number of movies that can be watched per each age based on the recommended age. For each target age, *Reducer* counts the movies from age 0 to the target age. However, *Mapper* uses incorrectly the recommended age of the movie as key, and the *Reducer* could not receive all movies from age 0 to the target age. The program fails when a *Reducer* receives the movies of one age but does not receive those of the lower ages due the parallelism done by the framework.

(P11) *studentGPA* calculates the GPA from several grades. The *Mapper* calculates the GPA of those grades locally available, and *Reducer* incorrectly emits them directly without aggregating them. The program fails when the framework splits the grades of the same student in several *Mappers*, and *Reducer* does not aggregate the partial GPAs.

(P12) *CarAccidents* calculates the average visibility level per severity type. The program has an *In-Mapper Combiner* followed by another *Combiner* incorrectly implemented with a non-commutative/non-associative functionality. The program fails when the framework executes the *Combiner*.

(P13) *weatherAnalysis* obtains a delta between the maximum and minimum snowfall per each state-month. The *Mappers* receive the snowfall sorted by state-month and for each one only emit the first and last snowfall. However, the *Reducer* incorrectly calculates the delta from the first and last snowfall of each state-month, even when they are not the maximum or minimum. The program fails when the snowfalls are unsorted due the parallel execution of the framework and *Reducer* is not aware of that.

The *population* of the experiments is composed by all test cases that trigger *MapReduce* design faults, and each of these test cases is taken as the *experimentation unit*. The test cases are generated randomly for each of the previous subject programs guaranteeing that they trigger a design fault. The experiments could be affected by the number of test input data, hence a *blocking factor* [48] is established with different numbers of $\langle key, value \rangle$ pairs varying between 10 and 99999 pairs. This means that the experiments are replicated with different blocks, each one indicates the lower and upper limit of number of $\langle key, value \rangle$ pairs and analysed individually to both control and detect some influence of the number of test input data in the debugging techniques. Hence, the test cases are not only generated randomly but also with the number of $\langle key, value \rangle$ pairs established in the *blocking factors* designed per each research question. The research questions related to MRDebug-FL, RQ1.1-RQ1.4 and RQ3.1-RQ3.2, have the following *blocking factors*: between 10 and 99 $\langle key, value \rangle$ pairs, and between 100 and 999 $\langle key, value \rangle$ pairs, respectively. The research questions that execute the input reduction, RQ2.1-RQ2.3 and RQ3.3 have the following *blocking factors*: between 10 and 99,

100 and 999, 1000 and 9999, and between 10000 and 99999 $\langle key, value \rangle$ pairs. The following subsections address the research questions repeating each experiment 30 times for each subject program and blocking factor. The *experiment environment* is a Linux server with 10GB of RAM, the subject programs are implemented using *Hadoop MapReduce*, and they are executed in local. The size of the test input data varies from 10 up to 99999 $\langle key, value \rangle$ pairs depending on the *blocking factor*.

Subsection VII-A addresses the research questions of MRDebug-FL, Subsection VII-B the RQs of MRDebug-IR, and Subsection VII-C the RQs of MRDebug-IR-FL. Finally, Subsection VII-D discusses the general results, and Subsection VII-E the limitations of the experiments. The supplemental material with the test cases and the scripts used in the evaluation are in Zenodo [49].

A. MRDebug-FL Evaluation

This section addresses the research questions RQ1.1-RQ1.4 by executing MRDebug-FL with the test cases and the program subjects described above. MRDebug is executed with $K = 5$ (we evaluate the impact of this parameter in RQ1.4) and using the 52 ranking metrics listed in the supplemental material: Ochiai1 and Tarantula, among others, that are the most common in the literature [18], [36], [37]. This section only shows the aggregated results with the Ochiai1 ranking metric due to the limitation in space. The results of each *blocking factor*, subject program or ranking metric are similar to those shown in this section, and they are provided as supplemental material.

In fault localization it is common to have ties in the ranking [50]. A tie occurs when MRDebug-FL assigns the same suspiciousness to multiple characteristics and they are simultaneously in the same ranking position. Since it is not possible to rank several characteristics in the same position as the tester can only analyse one of them at a time, the tester breaks the tie by randomly selecting the characteristics of the tie one by one. This randomness can have an impact on the results of the experiments when the root cause of the fault is tied with other characteristics because ranking one before the other is a matter of chance. To control this randomness during the evaluation of MRDebug-FL, the following three tie-breaking scenarios are applied: best, average and worst scenarios of each tie that contains the root cause of the fault. The best scenario breaks the tie by ranking the root cause of the fault above the other characteristics of the tie, i.e., the tester analyses the root cause of the fault before the other characteristics of the tie. In the same way, the worst scenario ranks it below the other characteristics of the tie i.e., the tester analyses the root cause of the fault after the other characteristics of the tie. Finally, the average scenario ranks it in the middle of the tie. In the evaluation of MRDebug-FL, we analyse the ranking of the potential root causes of faults (characteristic ranking) obtained by the fault localization technique, and execution time. Precisely, the *dependent variables* or *response variables* are: the position of the root cause of fault in the characteristic ranking, and execution time in seconds. The experiments answer the research questions using different statistical measures and qualitative analyses described below. Among the

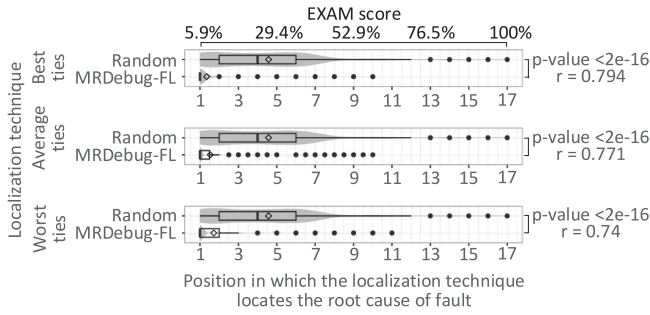


Fig. 10. Distribution of the position of the root cause of the fault by random localization and MRDebug-FL.

statistic metrics, the p-values and effect sizes are used to validate the hypotheses proposed in the research questions. The p-values indicate whether an effect exists in the hypothesis, and the effect sizes indicate how large that effect is. We consider that $p\text{-value} \geq 0.05$ indicates a non-statistical significance for the effect, whereas $p\text{-value} < 0.05$ indicates a statistical significance. In the case of the effect sizes, we use the Cohen effect sizes and the interpretation of Coolican [51]: $r > 0.5$ indicates large effect, $r > 0.3$ is medium effect, $r > 0.1$ is small effect, and $r < 0.1$ indicate no effect.

RQ1.1. Is MRDebug-FL more effective at locating the root cause of MapReduce design fault than a random localization (baseline)? To answer this research question we use the EXAM score [52] which is one of the most used evaluation metrics in fault localization [36], and the normalized AUC (Area Under Curve), because it allows comparison between ranking metrics [53]. The EXAM score measures the percentage of the characteristic ranking that must be examined until the position of the root cause of the fault is reached. The AUC considers per each test case the position of the root cause of the fault in the characteristic ranking, and it is defined as the sum per each test case of the percentage of characteristic ranking not analysed. We compare the characteristic rankings provided by both random localization (*baseline*) and MRDebug-FL using the non-parametric *statistic test* Wilcoxon Sign Rank test that measures the differences between the paired medians with the following one-tail null hypothesis: $H01$: The position of the root cause of the fault in the characteristic ranking obtained by MRDebug-FL is worse than or equal to that obtained by a random location.

Fig. 10 shows that MRDebug-FL is better than random localization in locating the root cause of the faults in the first positions of the characteristic ranking. The figure has three plots, one per each tie-breaking strategy, and each plot is the distribution of the positions in the characteristic ranking in which the fault is located by MRTest-FL and random localization for all programs and *blocking factors*. The Y-axis of each plot is the ranking metric used. The X-axis of each plot is the position of the fault in the characteristic ranking using both the position in the ranking and the EXAM score. Position 1 of the ranking means that the debugging technique correctly locates the fault in the best position, and the worst position is 17. In other terms, a debugging technique is more effective when the root cause of the fault is closer to position 1 of the characteristic ranking (X-axis). It can

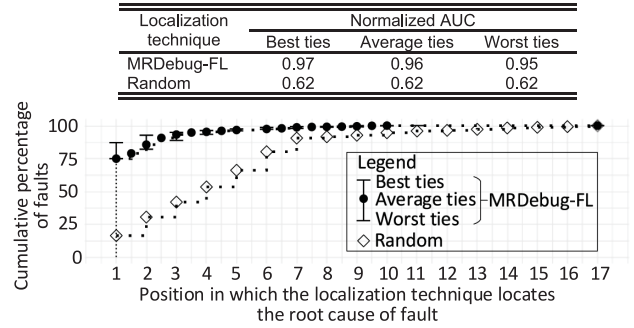


Fig. 11. Normalized AUC of MRDebug-FL.

be observed in the best tie-breaking strategy (top plot) that, in median, Random locates the faults in position 4 (position 4.6 in average), while MRDebug-FL locates them in position 1 (position 1.4 as average). In the three tie-breaking strategies, MRDebug-FL ranks the root cause of the fault in the first top positions, which corresponds to a low EXAM score. Fig. 10 also shows on the right the Wilcoxon Sign Rank p-values and Cohen effect sizes in comparing MRDebug-FL against a random localization. Regardless of the tie-breaking technique, we can observe that MRDebug-FL is better than random localization with a $p\text{-value} < 2e-16$ and large effect sizes with $r > 0.74$.

Fig. 11 compares MRDebug-FL with the random localization according to the normalized AUC. The X-axis is the position in which the localization technique locates the root cause of the fault, and the Y-axis is the cumulative percentage of faults located at each position for all programs and *blocking factors*. A debugging technique is better when the cumulative percentage of faults (Y-axis) is closer to 100 in the first positions (X-Axis) because this means that the technique locates more faults in those positions. It can be observed that Random locates 16.3% of faults in the first position while MRDebug-FL locates in this position 87.1% of faults with the best tie strategy and 74.9% in both average and worst ties. The cumulative number of faults localized in all positions is measured with the normalized AUC that is represented in the table on the top of the figure. A debugging technique is more effective when the normalized AUC is closer to 1. We can observe in the table that MRDebug-FL has a normalized AUC near to the maximum because it is greater than 0.95. In contrast, the random localization has 0.62 of normalized AUC. This means that MRDebug-FL not only locates the root cause of the fault in the top positions of the characteristic ranking, but it is also more effective than random localization.

Based on the above, the null hypothesis is rejected and we can state that MRDebug-FL is significantly better than random localization because it locates the root cause of the fault in the first positions of the characteristic ranking and with more normalized AUC.

RQ1.2. In what position of the suspiciousness ranking obtained by MRDebug-FL is the root cause of the MapReduce design fault? To answer this research question we use the $acc@n$ metric that measures how many times the root cause of the fault is localized before the N position. The comparison is done by the non-parametric *statistical test* Wilcoxon Sign Rank

Localization technique	P-values (H_{02})															Legend p-values: □ Non significant ■ Significant
	Positions best tie					Positions average tie					Positions worst tie					
	acc@1	acc@2	acc@3	acc@4	... acc@17	acc@1	acc@2	acc@3	acc@4	... acc@17	acc@1	acc@2	acc@3	acc@4	... acc@17	
MRDebug-FL	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	<2e-16	
Random	1	1	1	6e-13	<2e-16	1	1	1	6e-13	<2e-16	1	1	1	6e-13	<2e-16	

Localization technique	Effect sizes (H_{02})															Legend Effect sizes: □ No effect ($r < 0.1$) ■ Large effect ($r > 0.5$)
	Positions best tie					Positions average tie					Positions worst tie					
	acc@1	acc@2	acc@3	acc@4	... acc@17	acc@1	acc@2	acc@3	acc@4	... acc@17	acc@1	acc@2	acc@3	acc@4	... acc@17	
MRDebug-FL	0.948	0.948	0.948	0.948	0.948	0.916	0.916	0.916	0.916	0.916	0.916	0.916	0.916	0.916	0.916	
Random	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	0.868	

Fig. 12. Hypothesis testing of faults rightly located in each position of the ranking for all programs according to different ranking methods.

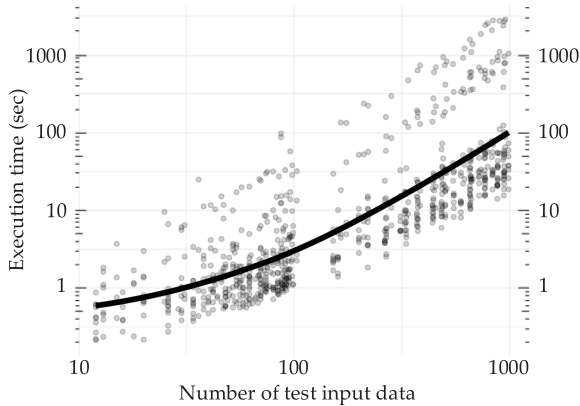


Fig. 13. Execution time trend of MRDebug-FL.

with the following one-tail null hypothesis: H_{02} : The position of the root cause of the fault in the characteristic ranking obtained by MRDebug-FL is greater than or equal to N (N is the position 1, 2 ...).

Fig. 12 has three sub-tables with the Wilcoxon Sign Rank p-values (top) and another three sub-tables with the Cohen effect sizes [54] (bottom) of each top N (acc@ N) position of the characteristic ranking for all programs in all *blocking factors*. Each one of the sub-tables is one of the tie-breaking techniques (best, average and worst) for random localization and MRDebug-FL with Ochiai1 ranking metric. The two rows of each table are MRDebug-FL with the ranking metric Ochiai1 and the random localization, and the columns are the top N (acc@ N) positions. A debugging technique is better when the first top positions -lower N in acc@ N - have both lower p-value -at least below 0.05- and larger effect sizes r . We can observe that MRDebug-FL localizes the fault in the first top positions of the ranking (Top 1) with a p-value < 0.05 (significant) and large effect sizes with $r > 0.5$. Therefore, the null hypothesis is rejected for the top positions of MRDebug-FL and we can **state that, in general, MRDebug-FL localizes the root causes of the fault in the first positions of the characteristic ranking.**

RQ1.3. How much execution time does MRDebug-FL employ to locate the MapReduce design fault? To answer this research question, we analyse the trend of the execution time. Fig. 13 shows the execution time of MRDebug-FL according to the number of $\langle key, value \rangle$ pairs of the test cases of all programs in all *blocking factors*. Each point is the execution of fault

localization in one test case, the X-axis is the number of the test input data, and the Y-axis is the execution time of MRDebug-FL. The curve depicted in the plot is the trend of the execution time and it grows according to the number of $\langle key, value \rangle$ pairs. As expected, the more test input data to debug, the longer execution time. Note that MRDebug-FL executes the unit test cases in one computer and not in the cluster. The test cases with less than 100 $\langle key, value \rangle$ pairs employ less than 100 seconds in the worst cases. However, those test cases with ~ 1000 $\langle key, value \rangle$ pairs that employ more execution time take approximately 2800 seconds (~ 47 minutes). The execution time also follows an exponential trend. Therefore, **we can state that MRdebug-FL employs low execution time when the test cases have few $\langle key, value \rangle$ pairs, but the execution time grows exponentially according to the number of the test input data.** Note that this execution time can be decreased by MRDebug-IR-FL that combines input reduction and fault localization to localize a fault only with part of the test input data (Subsection VII-C).

RQ1.4 Can MRDebug-FL achieve a trade-off between the design faults located and execution time by varying the number of times that each configuration is executed (parameter K)? Intuitively, a higher number of configurations executed by MRDebug-FL (parameter K) yield better results, but also more execution time. To answer this research question, we analyse the trend of the normalized AUC in the ranking metrics per each one of the following values of K : 1, 2, 3, 5, 8 and 13. For the best K , we also analyse the number of configurations that fail/pass.

Fig. 14 depicts the increased trend of both execution time of MRDebug-FL and the normalized AUC when more configurations are executed by MRDebug-FL. The figure has two plots, the top plot is the trend of the normalized AUC of MRDebug-FL with Ochiai1 ranking metric and the bottom plot is the execution time varying in both plots the number of times each configuration is executed. The Y-axis of the top plot is the normalized AUC and the Y-axis of the bottom plot is the execution time. In both plots, the X-axis is the K parameter, and the experiments are re-executed in each one of the K aggregating the data of all programs and all *blocking factors*. The best K parameter of MRDebug-FL (X-axis) is the one that achieves the best trade-off between effectiveness (Y-axis of top plot) and efficiency (Y-axis of bottom plot). This means that the best K is the position of X-axis that has both the normalized AUC closer to 1 (Y-axis of top plot) and lower execution time (Y-axis of bottom plot). As expected, we can observe that MRDebug-FL is

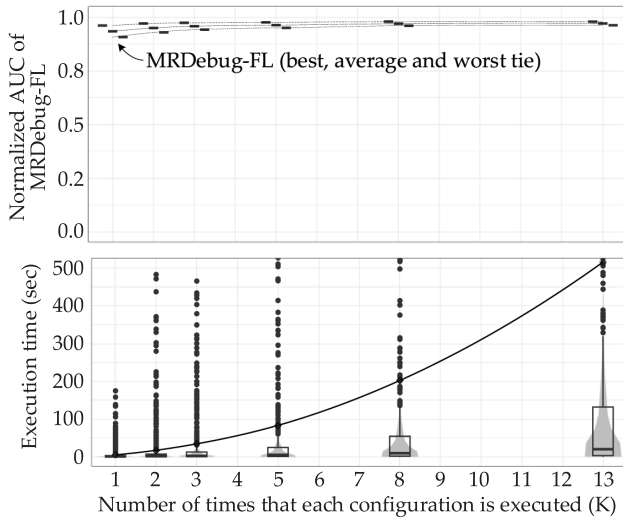


Fig. 14. Normalized AUC and execution time according to the K by MRDebug-FL.

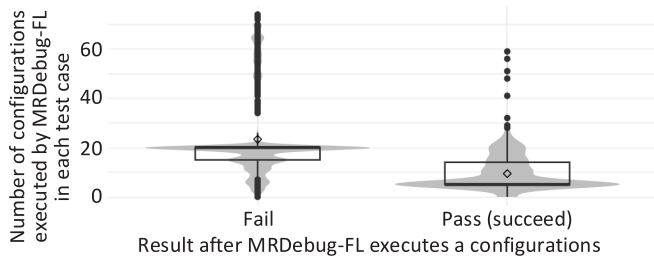


Fig. 15. Number of configurations that fail and pass per each execution of MRDebug-FL.

more effective when the configurations are executed more times, but with decreased efficiency. There is no value of K that achieves the best effectiveness and efficiency at the same time. From $K = 1$ to $K = 5$ the normalized AUC is improved between 2% and 4%, depending on the tie-breaking technique, whereas the execution time is slightly increased. In contrast, from $K = 5$ to $K = 13$ the execution time increases faster, and the normalized AUC only increases between 0% and 1%. Therefore, we can state that $K = 5$ achieves a good trade-off between effectiveness and efficiency.

Fig. 15 depicts the distributions of the number of configurations that fail and succeed during the execution of MRDebug-FL with $K = 5$. Per each test case, MRDebug executes several configurations where some of them succeed, and others fail. Note that all of these configurations (succeeded or failed) are useful to locate the root cause of the fault. It can be observed that, in average, MRDebug-FL executes per each test case 23.4 configurations that fail and 9.4 that succeed. However, this is not always true for all programs because we have observed in the supplemental material that in some programs it is just the opposite. We cannot conclude neither that MRDebug-FL executes more configurations that succeed than that fail, nor the vice versa. Regardless, MRDebug-FL usually executes per each test case a variety of configurations that both fail and succeed.

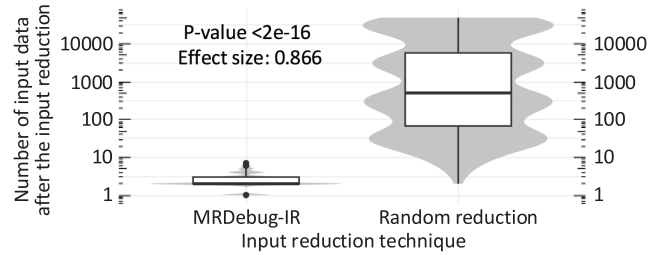


Fig. 16. Distribution of the test input data after the reduction.

B. MRDebug-IR Evaluation

This section addresses the research questions RQ2.1-RQ2.3 by executing MRDebug-IR with the test cases and the program subjects described at the beginning of this section. In the evaluation of MRDebug-IR, we analyse the test input data after the reduction, and the execution time. The *dependent variables* or *response variables* are: the number of the $\langle \text{key}, \text{value} \rangle$ pairs after the reduction, and the execution time in seconds. This section only shows the aggregated results due to the limitation of space, but the results of each *blocking factor* or subject program are provided as supplemental material.

RQ2.1 Is MRDebug-IR more effective at isolating the test input data that trigger the MapReduce design faults than a random reduction (baseline)? To answer this research question, we analyse the number of test input data after the reduction of MRDebug-IR and the random reduction (*baseline*). This random reduction makes random searches in the space as much time as MRDebug-IR, that is, the technique selects randomly without substitution several subsets of the test input data employing the same time as MRDebug-IR. The comparison is done by the non-parametric *statistical test* Wilcoxon Sign Rank with the following one-tail null hypothesis: H_0 : The number of test input data after the reduction of MRDebug-IR is greater than or equal to that after random reduction.

Fig. 16 depicts the distribution of the number of test input data after the reduction done by both MRDebug-IR and random reduction (*baseline*) for all programs in all *blocking factors*. The X-axis is the input reduction technique used, and the Y-axis is the number of the test input data after the reduction. A debugging technique is better when the number of test input data after the reduction (Y-axis) is closer to 1. The figure also shows the Wilcoxon Sign Rank p-value and Cohen effect size that compare MRDebug-IR against a random reduction. MRDebug-IR is more effective than the baseline when both the p-value is lower -at least below 0.05- and r is larger. In these experiments, MRDebug-IR always reduces the data below 10 $\langle \text{key}, \text{value} \rangle$ pairs, whereas the random reduction works worse. We can observe that MRDebug-IR is significantly better than the random reduction with a p-value $< 2e-16$ and large effect sizes with an $r = 0.866$. Therefore, the null hypothesis is rejected, and we can state that MRDebug-IR is significantly better at reducing data than random reduction.

RQ2.2 How effective is MRDebug-IR in reducing the test input data of the MapReduce design faults? To answer this research question, we analyse the differences in the number of test input data before and after MRDebug-IR. We calculate the

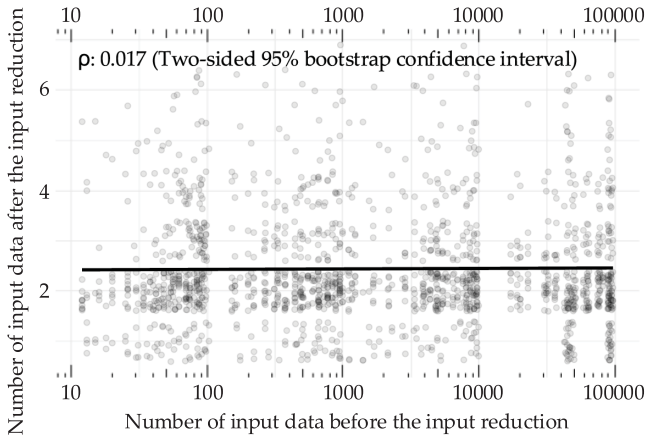


Fig. 17. Trend of test input data before and after MRDebug-IR.

average of the number of the test input data after the reduction, and conduct a bivariate correlation analysis between the number of the test input data before and after the reduction.

Fig. 17 shows the relationship between the number of test input data before and after the reduction. The figure depicts one point per each test case indicating the number of test input data before (X-axis) and after (Y-axis) MRDebug-IR for all programs and *blocking factors*. The trend depicted in the figure indicates how many test input data are obtained after the reduction (Y-axis) according to the test input data before the reduction (X-axis). The debugging technique is better when the trend is closer to 1 in the Y-axis because this means that the debugging technique reduces many more data. In the same way, when the trend is horizontally flat it means that the debugging technique always obtains the same number of test input data after the reduction (Y-axis) regardless of how large the test input data were before the reduction (X-axis). The figure also indicates the Spearman's correlation coefficient, ρ , which indicates the strength -if any- in the relationship between the number of input data before the reduction and after the reduction. The strength of relation is bigger when ρ is closer to 1. MRDebug-IR reduces the data until reaching 2.5 *<key, value> pairs* as mean, which is optimal or close to the optimal. We can observe that MRDebug-IR reduces the data constantly around 2.5 *<key, value> pairs* regardless of how much bigger the test input data are before the reduction. There is no correlation between the number of test input data before and after MRDebug-IR. The Spearman's correlation is near to 0 (0.017) obtained with two-sided 95% bootstrap confidence interval. The reduction is independent from how much data are in the test case, and it is also near to the optimal reduction. Therefore, **we can state that MRDebug-IR is both effective and reliable** because it reduces the test input data near to the optimal.

RQ2.3 How much execution time does MRDebug-IR employ to reduce the test input data of the MapReduce design faults? To answer this research question, we analyse the trend of the execution time. Fig. 18 shows the execution time of MRDebug-IR according to the number of *<key, value> pairs* of the test cases for all programs in all *blocking factors*. Each point is the execution of one test case and the curve is the trend

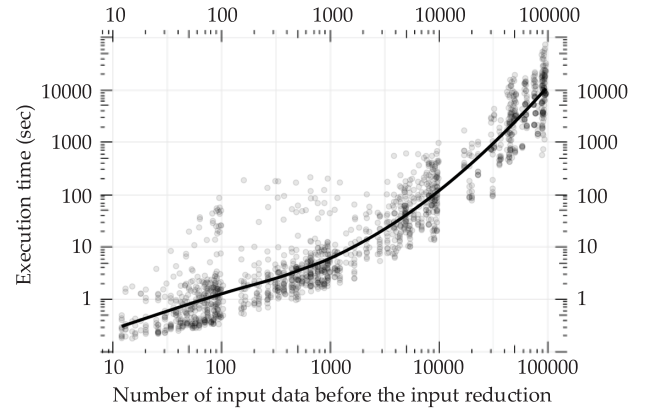


Fig. 18. Execution time trend of MRDebug-IR.

between the number of test input data before the reduction (X-axis) and the execution time (Y-axis). As expected, when the more the test input data to debug, the longer the execution time. MRDebug-IR reduces the test cases with 1000-9999 *<key, value> pairs* in 86.7 seconds on average. However, if the test case has 10000-99999 *<key, value> pairs* the execution time is increased until 6975 seconds (~ 2 hours) on average. Therefore, **we can state that MRDebug-IR employs low execution time when the test cases have few *<key, value> pairs*, but the execution time increases exponentially**. Note that MRDebug-IR executes the unit test cases in one computer and not in the cluster in order to fine-grain control the execution.

C. MRDebug-IR-FL Evaluation

This section addresses the research questions RQ3.1-RQ3.3 by executing both MRDebug-IR-FL and MRDebug-FL with the test cases and the program subjects described in the beginning of this section. The *baseline* of RQ3.1 and RQ3.2 is MRDebug-FL, and the *treatment* is MRDebug-IR-FL which is configured to localize the faults after reducing the test input data down to the minimum possible. In RQ3.3, we analyse how effective/efficient MRDebug-IR-FL is at locating faults when it is configured to reduce more or less data. In all the research questions, the MRDebug techniques uses $K = 5$ because, according to RQ1.4, this achieves a good trade-off between effectiveness and efficiency. As we can observe in Subsection VII-A, MRDebug-FL does not scale well, and to make a fair comparison between MRDebug-FL and MRDebug-IR-FL, in RQ3.1 and RQ3.2 we use the *blocking factors* 10-99 and 100-999 *<key, value> pairs*. In contrast, RQ3.3 only analyses MRDebug-IR-FL and we use the following *blocking factors*: between 10 and 99, 100 and 999, 1000 and 9999, and between 10000 and 99999 *<key, value> pairs*. This section only shows the aggregated results with the Ochiai1 ranking metric due to the limitation of space, and the supplemental material contains the results of each *blocking factor*, subject program or ranking metric.

RQ3.1 Is MRDebug-IR-FL more effective at locating the MapReduce design faults than MRDebug-FL? To answer this research question, we analyse the position of the fault in the characteristic ranking provided by both MRDebug-FL and MRDebug-IR-FL. The comparison is done using the non-

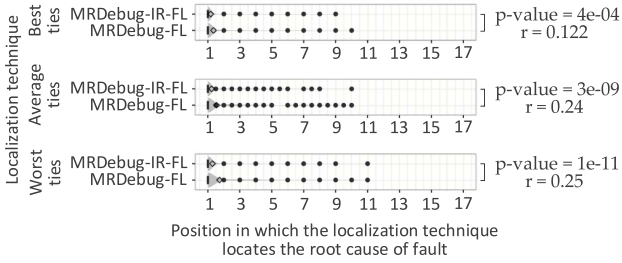


Fig. 19. Distribution of the position of the root cause of the fault by MRDebug-IR-FL and MRDebug-FL.

parametric *statistical test* Wilcoxon Sign Rank with the following one-tail null hypothesis: H04: The position of the root cause of the fault in the characteristic ranking obtained by MRDebug-IR-FL is worse than or equal to that obtained by MRDebug-FL.

Fig. 19 depicts how much better MRDebug-IF-FL is than MRDebug-FL at locating the root cause of the faults in the first positions of the characteristic ranking. The figure has three plots, one per each tie-breaking strategy, and each plot depicts for all programs and *blocking factors* the distribution of positions of root cause of fault in both MRDebug-FL and MRDebug-IR-FL. The X-axis of each plot is the position of the fault in the characteristic ranking, and the Y-axis of each plot the debugging techniques MRDebug-IR-FL and MRDebug-FL. A debugging technique is more effective when the root cause of the fault is closer to position 1 of the characteristic ranking (X-axis). The figure also shows the Wilcoxon Sign Rank p-values and Cohen effect sizes that compares MRDebug-IR-FL against MRDebug-FL. The debugging technique MRdebug-IR-FL is more effective than MRDebug-FL when regardless of the tie-breaking technique both the p-value is lower -at least below 0.05- and r is larger. Note that MRDebug-FL localizes the faults without reducing the test input data, and MRDebug-IR-FL localizes after reducing the test input data as much as MRDebug-IR can. It can be observed in the best tie-breaking strategy (top plot) that, in average, MRDebug-FL locates the faults in the position 1.4, while MRDebug-IR-FL locates them in position 1.2. Although MRDebug-FL is able to locate the root cause of the faults in the first positions of the characteristic ranking, MRDebug-IR-FL locates the faults even in higher positions. Regardless of the tie-breaking technique, we can observe that MRDebug-IR-FL is better than MRDebug-FL with a p-value < 0.0001 and small-medium effect sizes $r > 0.122$. Therefore, the null hypothesis is rejected, and **we can state that MRDebug-IR-FL is significantly better than MRDebug-FL** because it locates the root cause of the fault in better positions of the characteristic ranking.

RQ3.2 Is MRDebug-IR-FL able to decrease the execution time of MRDebug-FL? To answer this research question, we analyse the execution time employed by MRDebug-IR-FL and MRDebug-FL. We compare the execution time using the non-parametric *statistical test* Wilcoxon Sign Rank with the following one-tail null hypothesis: H05: The execution time of MRDebug-IR-FL is greater than or equal to that of MRDebug-FL.

Fig. 20 depicts for all programs and *blocking factors* the distribution of the execution time of both MRDebug-FL, and MRDebug-IR-FL locating the faults after reducing the test input

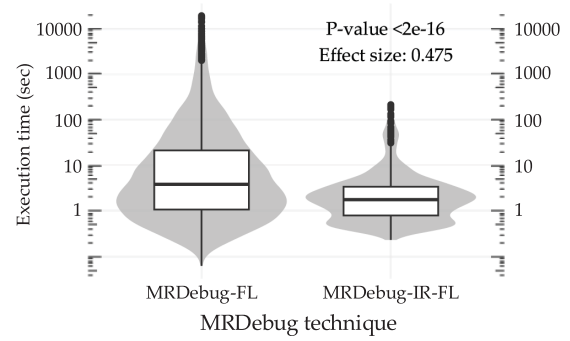


Fig. 20. Distribution of execution time in both MRDebug-FL and MRDebug-IR-FL.

until the maximum possible. The Y-axis is the execution time, and the X-axis the debugging techniques MRDebug-IR-FL and MRDebug-FL. The figure also shows the Wilcoxon Sign Rank p-values and Cohen effect sizes of the comparison. MRDebug-IR-FL is more efficient than MRDebug-FL when both the p-value is lower -at least below 0.05- and r is larger. We can observe that, in median, MRDebug-FL 3.8 seconds to locate the fault while MRDebug-IR-FL employs 1.8 seconds to both reduce the data and locate the fault. In the worst case, MRDebug-FL locates the fault in ~ 19000 seconds (5.3 hours) while MRDebug-IR-FL only employs ~ 200 seconds in the worst case to both reduce the data and locate the fault. MRDebug-IR-FL is significantly faster than the MRDebug-FL with a p-value $< 2e-16$ and medium/large effect sizes with an $r = 0.475$. Therefore, **we can state that MRDebug-IR-FL is faster than MRDebug-FL at localizing the faults** because the input reduction of MRDebug-IR-FL removes all irrelevant data and the localization is done with less data.

RQ3.3 How much data should MRDebug-IR-FL reduce to localize the faults in both a more effective and more efficient way? To answer the research question, we analyse the normalized AUC and the execution time employed by MRDebug-IR-FL. Depending on how many test input data MRDebug-IR-FL reduces, the localization can yield different results and execution time. We execute the test cases forcing MRDebug-IR-FL to reduce the data down to the following reduction thresholds: 10, 500, 1000 and 1500 $\langle key, value \rangle$ pairs.

Fig. 21 shows the normalized AUC for MRDebug-IR-FL varying the reduction threshold (i.e., how many test input data are after the reduction) in all programs and *blocking factors*. The X-axis is the reduction threshold, and the Y-axis is the normalized AUC obtained by the fault localization done by MRDebug-IR-FL. The most effective reduction threshold (X-axis) for MRDebug-IR-FL is the one that achieves the best normalized AUC (Y-axis) closer to 1. When the data is reduced down to 1500 $\langle key, value \rangle$ pairs, the normalized AUC is high, between 0.93 and 0.97, depending on the tie-breaking technique, but when MRDebug-IR-FL reduces down to 10 $\langle key, value \rangle$ pairs, the normalized AUC is increased until it reaches between 0.96 and 0.97, depending also on the tie-breaking technique. In general, we can observe that MRDebug-IR-FL increases progressively the normalized AUC when there are less data to

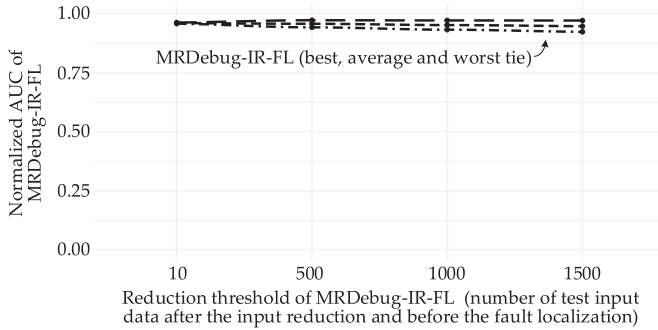


Fig. 21. Normalized AUC according to the reduction thresholds by MRDebug-IR-FL.

localize. This means that MRDebug-IR-FL is better when the data is reduced down to the minimum possible.

Fig. 22 depicts the total execution time of MRDebug-IR-FL varying the reduction threshold. The Y-axis is the total execution time employed by MRDebug-IR-FL, and the X-axis is the number of $\langle key, value \rangle$ pairs of the test input data before MRDebug-IR-FL, i.e., before the input reduction. In this figure we use different colours for each different reduction threshold. Note that MRDebug-IR-FL reduces the test input data down the threshold and localizes the root cause of the fault with the reduced data. Depending on the reduction threshold used, MRDebug could be more efficient. The best reduction threshold in terms of efficiency is the one that has the curve trend lower and closer to 0. As expected, we can observe that MRDebug-IR-FL employs different execution time depending on the reduction threshold. For example, in a test case with 2000 $\langle key, value \rangle$ pairs, if MRDebug-IR-FL reduces the test input data down to 1500 $\langle key, value \rangle$ pairs, it employs in total ~ 650 seconds on average, but if the reduction is down to 10 $\langle key, value \rangle$ pairs the total execution time of MRDebug-IR-FL decreases by employing ~ 15 seconds on average. In general, regardless of the test input data, we can observe -as can be expected- that MRDebug-IR-FL employs less execution time when there is less data to localize. This means that MRDebug-IR-FL is faster when the data is reduced down to the minimum possible.

Therefore, **we can state that MRDebug-IR-FL localizes the faults both better and faster when it reduces the data to the maximum possible degree** because the input reduction removes the irrelevant data.

D. Discussion of Results

The experiments indicate that the *MapReduce* design faults can be automatically located and the test input data can be automatically reduced until close to the maximum reduction. In the remainder of this subsection, the results of each debugging technique are discussed.

MRDebug-FL: the results of RQ1.1 and RQ1.2 (Subsection VII-A) indicate that when using the Ochiai1 ranking metric, MRDebug-FL locates automatically the right root cause of the faults in the first positions of the characteristic ranking and does so significantly better than random localization. In the supplemental material we also analyse MRDebug-FL using 52 ranking

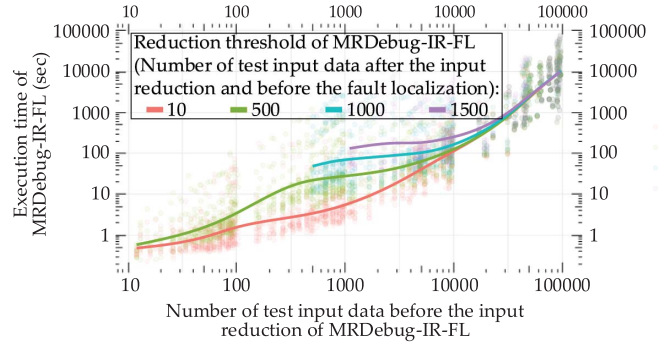


Fig. 22. Execution time for different reduction thresholds in MRDebug-IR-FL.

metrics and the results are similar to when using Ochiai1. All of the ranking metrics are significantly better than random localization in best and average ties. In the worst ties, 51 out of 52 ranking metrics are significantly better than random localization. The majority of ranking metrics also achieve a normalized AUC greater than 0.9 in all ties. The best ranking metrics in the experiments are Kulczynski2, McCon, and M2 that achieve the following normalized AUC values: 0.98 in best ties, 0.96 in the average ties, and 0.95 in the worst ties. Almost all ranking metrics also locate the fault in the first positions (Top 1) with p -value < 0.05 in best and average ties. In the worst ties, 40 out of 52 ranking metrics locate the fault in the top 1 ($acc@1$) with p -value < 0.05 , and the majority of them (46 out of 52) locate the fault in top 2 ($acc@2$) with p -value < 0.05 .

The results of RQ1.4 indicate that MRDebug-FL achieves a trade-off between effectiveness and efficiency when it locates the fault executing each configuration 5 times (parameter $K = 5$). We also observe that MRDebug-FL obtains better effectiveness when the number of times that each configuration is executed increases. These results have similarities with those obtained by Abreu et al. [55] for general-purpose programs. In both cases, *MapReduce* design faults and general-purpose faults, increasing the number of runs analysed by the fault localization improves its effectiveness.

In general terms, MRDebug-FL executes several configurations that fail and other configurations that succeed. The distribution of them depends on the program under debugging, but they are enough to locate the root cause of the fault. These results also have similarities with those obtained by Abreu et al. [55] for general-purpose programs: executing few configurations that fail is enough to obtain a near-optimal effectiveness of the debugging technique.

MRDebug-IR: isolates automatically the data that trigger the fault achieving the maximum reduction or near to the maximum reduction possible. MRDebug-IR employs low execution time with small test input data, but the execution time increases rapidly. Note that MRDebug-IR uses Delta Debugging, and other authors also consider that the execution time of Delta Debugging can grow exponentially. According to Misherghi et al. [56], the input reduction of AST (Abstract Syntax Tree) is NP-Complete, and Kalhauge et al. [57] prove that the input reduction problem is also NP-Complete in a broader way for dependency graphs. The search space of input reduction is 2^n

and the complexity of Delta Debugging is $O(n^2)$ in the worst cases according to Zeller et al. [20], where n is the size of the test case setting (number of $\langle \text{key}, \text{value} \rangle$ pairs in our domain). For example, while Delta Debugging can take up to 3 hours to reduce a test case with $n = 14092$ in a general purpose program [58], MRDebug-IR spends on average less than 7 minutes in the experiments for the same n but in *MapReduce* programs.

MRDebug-IR-FL: despite the fact that both techniques, MRDebug-FL and MRDebug-IR, are effective, the execution time increases quickly according to the number of the test input data, especially in MRDebug-FL. The execution time of fault localization can be decreased if the test input data is reduced before the localization. This means that instead of localizing the fault with all the test input data, the input reduction technique removes the irrelevant data and, after that, the fault localization technique locates the fault with only the small part of the test input data that is relevant. This is done by MRDebug-IR-FL, that executes the localization after the input reduction. MRDebug-IR-FL improves not only the execution time but also the effectiveness of fault localization (MRDebug-FL). In the supplemental material we also analyse MRDebug, using the 52 ranking metrics and in almost all of them MRDebug-IR-FL is significantly better than MRDebug-FL for best, average and worst ties. In the best ties, MRDebug-IR-FL is significantly better than MRDebug-FL in 50 out of 52 ranking metrics, and in the case of both average and worst ties, it is better in 48 out of 52 ranking metrics. Note that MRDebug-IR-FL not only localizes the faults both better and faster than MRDebug-FL, but in less execution time it also obtains a minimal data that triggers the failure.

E. Limitations

MRDebug does not execute the program under test in a cluster, but in a local environment so to guarantee both a fine-grain control and a reproducibility that is not easy to achieve in *Big Data* clusters due to the parallel non-deterministic executions. However, this local execution inhibits debugging test cases with large production datasets, as it is also the case with the unit testing tools commonly used in *Big Data* such as MRUnit or other XUnit tools that are executed locally. The remainder of this subsection discusses the limitations of the experiments through the threats of validity and their subcategories [48], [59], [60].

The **conclusion** threats are those issues that could affect the conclusions drawn from the experiments. The MRDebug techniques of this paper take advantage of MRTTest to automatically check if there is a failure or not. MRTTest could erroneously indicate to MRDebug that one execution succeeds when in fact it should indicate failure (*inaccurate data*). Although these errors could decrease the effectiveness of MRDebug, MRTTest provides the right information on average 60% to 100% of the time, depending on the program and test input data [14]. This is not a major issue because even with these inaccurate data, MRDebug is effective enough, as can be seen in the experiments.

The **internal** threats are those issues regarding the causal relationship between independent variables and dependent variables. RQ1.3, RQ2.3 and RQ3.2 analyse the execution time, but

some noise may be introduced into the measurements by other operative system tasks (*confounding effects of variables*). To mitigate this problem, the experiments were executed in the same computer without any other programs operating in the background.

The tool that automates MRDebug can contain faults and other limitations. To mitigate the potential faults of the tool, thorough manual/automatic testing was performed mainly from the functional and performance point of view.

The **external** threats are those issues that can affect the generalization of the results. The *experimentation units* are test cases randomly selected from a set of *MapReduce* programs. These programs were selected by purposive sampling. Ideally, the programs should also be selected randomly, but often this is not always feasible in software engineering [48] (*Interaction of selection and treatment*). For *Big Data* programs, there is no benchmark of design faults, and industrial programs are not usually available [61]. This problem was mitigated by using both faulty real-world programs and injecting a design faults in the *MapReduce* benchmarks used in the literature [40], [41]. The faults injected in the experiments (hand-seeded faults) could decrease the reproducibility of the experiments because seeded faults can usually be considered both subjective and not representative of real faults in terms of easy detection [62]. However, in our experiments we have observed that the results are similar in hand-seeded faults than in real-world faults.

The test cases used to evaluate MRDebug have up to 99999 $\langle \text{key}, \text{value} \rangle$ pairs. The results of the experiments could not be generalizable for larger sizes of the test input data (*Applicability of results across different samples*). To reduce the threat, the experiments analyse the relationships and trends according to the number of $\langle \text{key}, \text{value} \rangle$ pairs. Note that MRDebug is not executed in a cluster, and it is not able to debug test cases with large production data. According to other authors, debugging *Big Data* applications at scale have some fundamental obstacles [63], and executing several times the application at scale is also prohibitive expensive [64]. However, MRDebug does not intend to debug with production data, but with test data in development environment.

Other results can be obtained if MRDebug debugs the programs in smarter ways, that is if MRDebug-FL generates/analyses the configurations in a different way, or if MRDebug-IR employs a better search strategy (*Applicability of results when technique is varied*). In this regard, it is worth mentioning that MRDebug techniques achieve very good effectiveness, and the results of the experiments are reliable across different settings. In the case of MRDebug-FL, similar results are achieved in the majority of 52 different ranking metrics commonly used in the research. In the case of MRDebug-IR, the results show optimal or near to optimal effectiveness regardless of the test input data. In the case of MRDebug-IR-FL, the evaluation analyses different reduction thresholds, and the trends are consistent.

In the experiments, the techniques MRDebug-FL and MRDebug-IR are not compared with other state-of-the-art debugging techniques. None of the state-of-the-art debugging techniques is able to locate the root cause of the design faults or reduce the test input data automatically without a user-defined

oracle. In the case of the fault localization, these techniques are not able to obtain the characteristic that triggers the fault because they obtain suspicious lines of code. Another difference is that MRDebug obtains the root cause of the fault with only one test case, while the state-of-the-art fault localization techniques usually need several test cases. MRDebug-FL is a tailoring of the state-of-the-art fault localization techniques focused on the design faults. On the other hand, MRDebug-IR is also an adaptation of *Delta Debugging* to reduce the test input data of *MapReduce* design faults using MRTTest [14] as oracle. The other state-of-the-art input reduction techniques like data provenance [40], [65] usually require a user-defined oracle per each record of the outcome, while instead the oracle MRTTest judges the whole outcome. Therefore, neither MRDebug-FL nor MRDebug-IR can be compared with the state-of-the-art techniques. To reduce the threat, MRDebug-FL compares 52 ranking metrics and it is evaluated against a random localization, and MRDebug-IR against random reduction. The evaluation of MRDebug-IR-FL is done using MRDebug-FL as a baseline. The debugging techniques of MRDebug not only are better than the random baselines, but in absolute terms locate the faults in the best positions and reduce the data near to the maximum possible.

The **construct** threats are those issues between the experiment and its underlying theoretical concepts. MRDebug-FL and MRDebug-IR are only compared against a random localization/reduction because the other techniques of the literature are not suitable for *MapReduce* design faults. In the case of fault localization, the other techniques are usually focused on the analysis of the statements instead of on configurations. In the case of input reduction, the other techniques of the literature do not support the reduction of *MapReduce* design faults.

One part of the experiment analyses the efficiency of MRDebug techniques based only on the execution time measure, but there could be more measures not considered here, such as memory (*Mono-operation bias*). To mitigate this problem, the tool that automates the research was tested to avoid memory bottlenecks.

VIII. RELATED WORK

Debugging distributed programs is a difficult task, especially in the *Big Data* field [66]. Several works propose debugging techniques focused on performance for *Big Data* frameworks [67], [68] and others for the *MapReduce* programs [69], [70]. In contrast, the current paper does not focus on performance debugging, but on functional debugging. From practical point of view, functional debugging can be done manually or automatically, analyzing the code statically or dynamically, and also with runtime production data or with test data. There are some fundamental obstacles to debugging *Big Data* programs at scale in production [63]. Several debugging techniques require to instrument the code or analyze several executions in controlled way, but this is prohibitively expensive with large production data. Executing both testing and debugging in the development environment can be more practical than in production because the tests do not impact production, the environment is more

controllable, and there is no need for importing the test data into a distributed filesystem, among other advantages.

Olston et al. [71] interview ten employers of Yahoo! about debugging dataflow programs like *MapReduce*. The majority of them suggest that it can be valuable to obtain the data and operators that cause the failure. The current paper undertakes both tasks in *MapReduce* design faults through the MRDebug framework. MRDebug locates the root cause of the fault and isolates/reduces the data that trigger the failure. This debugging framework is executed automatically analyzing the test data and executing the code dynamically in development environment to handle accurately the whole execution.

Fault localization: Gecer et al. [72] investigate the debugging techniques used by seven *Big Data* developers and discover that these developers find the root cause of the faults by manual analysis of the logs, among others. Daphne [63] is a debugger for DryadLINQ (framework that supports and extends the *MapReduce* processing model). This debugger diagnoses the root cause of the faults based on a decision tree at different levels of abstraction considering logs and stack traces of the execution. The current work, MRDebug, analyses neither logs nor stack traces because it is focused on the failures that are triggered by some non-deterministic executions. Then MRDebug analyses with spectrum-based fault localization not only one execution, but it executes the program several times to locate the non-deterministic characteristics that trigger the failure.

Gulzar et al. [41] propose OptDebug that is a spectrum-based fault localization approach to locate the faulty line/operation in dataflow applications like *Spark*. OptDebug analyses which lines and operations (e.g., split of string, or minus operation between two integers) are covered during the test execution and if the executions fail/succeed. As result, OptDebug obtains a ranking of lines/operations that are more suspicious to contain a fault. MRDebug also uses spectrum-based fault localization for *Big Data* applications, but the kind of faults localized in OptDebug and MRDebug are different. OptDebug locates faults caused by an operation code, while MRDebug locates faults that are not caused by the code itself, but by the program design. MRDebug analyses execution patterns and does not analyse the lines/operations covered because, in case of design faults, the same coverage in the same program with the same test input data can sometimes succeed and other times fail due to the non-deterministic execution. Another difference is that OptDebug requires a user-defined oracle per each record of the outcome, whereas in contrast MRDebug does not require that the tester provides the oracle because it uses MRTTest [14]. OptDebug cannot use MRTTest as oracle because MRTTest is an oracle for the whole outcome and not for each record, and MRTTest is also focused on design faults instead of other faults. Both techniques, OptDebug and MRDebug, are complementary because they obtain different information about the fault: OptDebug obtains the faulty line of code and MRDebug the execution pattern that trigger the failure. In the case of general-purpose faults, it could be more useful to locate the line of code that trigger the failure. However, in the case of design faults, the failures are triggered by non-deterministic executions of the framework, and it could be more useful to locate which have in common the executions that triggers the failure.

Isolation/reduction of the data: Some *Big Data* developers reduce, re-create or ignore data manually to observe the faults [72], but there are some research works to automatize this task. BigSift [64] is a runtime debugger for applications executed in *Spark* (framework that supports and extends the *MapReduce* processing model). This debugger isolates the data through Delta Debugging combined with data provenance and a user-defined oracle per each record of the outcome. The current paper, MRDebug, also isolates the data based on Delta Debugging, but does not require that the tester provides an oracle because it uses MRTest [14]. Note that BigSift cannot use MRTest as oracle because MRTest only works as oracle for the whole outcome and not per each record. MRDebug is not focused on any type of faults, but it is focused only on design faults. During the execution of the Delta Debugging algorithm, MRDebug executes several times the application in a controlled environment, and it also allows to stop the reduction when a threshold is reached.

Feng et al. [73] propose to debug the *Big Data* applications by means of a binary search aimed to reduce the input data while at the same time preserving the test coverage. MRDebug is not focused on general-domain failures as the previous technique, but on the design faults of the *MapReduce* applications. Thus, MRDebug preserves the occurrence of the failure in any configuration instead of the test coverage.

Other debugging utilities: Breadcrumb [74] debugs the *Big Data* queries that do not produce a result, and the tool provides a trace of the reasons of the unexpected result. DPLOG [75] also provides intermediate traces together with state information about all the executions of the *Big Data* program. Inspector Gadget [71] is a debugger that alerts about predicate violations and also traces the data that produce the failures in Pig programs (high level language compiled as *MapReduce* program). Our previous work also alerts of potential failures in production [76], but only for those caused by *MapReduce* design faults. The current work, MRDebug, also allows to trace the failures, but only for those caused by design faults and only at a high level, executing the configuration that triggers the failure with breakpoints and watchpoints.

In the case of runtime debugging, Amber [77] is a *Big Data* platform that enables several debugging functionalities such as pauses during the program execution or conditional breakpoints, among others. Amber is based on the Actor model instead of the *MapReduce* processing model but achieves similar performance to *Spark*. According to one study [78] of Stack Overflow, one of the challenges of *Spark* is the lack of debugging tools that show the data processing details without significant runtime overhead. Another runtime debugger of *Spark* is BigDebug [79] that allows to insert simulated breakpoints and watchpoints directly in a production environment. Similarly, IDRA MR [80] supports breakpoints in the *MapReduce*-style cluster during the runtime, but the debugging is performed in a different machine based on a copy of the execution state. This type of out-of-place debugging is also done by Snoopy [81] in the *Spark* applications, supporting programmatic breakpoints and pausing, among others. In contrast to the previous techniques, the current paper, MRDebug, does not support breakpoints in production but simulates

these production environments to allow the insertion of the breakpoints and watchpoints in the configurations. Other works are focused on record and replay failures. Arthur [82] is a debugger for *Hadoop* and *Spark* that traces the relevant data and allows to replay the failure. Newt [83] is another debugger of *MapReduce* applications that captures runtime information allowing the tracing and reproduction of failures. Bergen et al. [84] propose a debugger for *Spark* that records failures from production and reproduces these failures locally to support breakpoints. The current work, MRDebug, is focused on non-deterministic failures, but the previous record-replay techniques do not handle these kinds of faults properly. Arthur [82] considers a checksum of the output and can then detect non-determinism, but is not able to reproduce non-deterministic results. Newt [83] can also record the non-deterministic data but is not able to reproduce them deterministically. The current work, MRDebug, not only captures the non-deterministic executions that cause failures, but also reproduces them deterministically through seeds.

Another way to reproduce the non-deterministic faults is forcing the test case execution to trigger the failure every time. However, this is challenging in *Big Data*. Despite the fact the tester can tune the framework parameters, the tester does not have fine-grain control of how the test case is executed because “some parameters interact subtly with the rest of the framework and/or job-configuration and is relatively more complex for the user to control finely” [34]. *Hadoop MapReduce* has more than 190 parameters that can affect the execution behaviour [33], and in each release an average of 4.14 parameters are created, 1.6 renamed and 0.16 removed [85]. The framework *Spark* also has a lot of parameters that can affect the execution behaviour, and almost 10% of *Spark* commits add/modify/remove parameters [86]. In addition to the number of parameters, the dependencies between other parameters are prevalent and diverse, and their handling is often deficient and ad hoc [87]. If the tester indicates 4 number of *Mappers*, the framework considers it as a hint and can execute a different number of *Mappers* regardless of the tester’s choice [34]. All of these parameters make it difficult in practice to debug the *Big Data* programs directly in production. In contrast, MRDebug executes the debugging in a controllable environment using simulation.

IX. CONCLUSION AND FUTURE WORK

This paper presents a debugging framework called MRDebug that automates both the fault localization and input reduction of *MapReduce* design faults. These faults are located using a spectrum-based fault localization technique, and the test input data is reduced using Delta Debugging, in both cases adapted to debug the *MapReduce* design faults. The experiments in both real-word and seeded programs show that the faults can be localized analysing only a few executions, and the test input data is reduced until the minimal or near to the minimal needed to trigger the fault.

In conclusion, MRDebug can help testers/developers understand the *MapReduce* design faults and the potential risk of running the faulty program in production. Faults such as these are

difficult to reproduce because they usually manifest themselves non-deterministically over a distributed framework that cannot be fine-grained controlled. MRDebug automatically obtains the root cause of the faults, making it easy to comprehend which part of the program is faulty and the circumstances that trigger/mask the fault. The tester/developer can obtain insights about the fault using MRDebug to reproduce the non-deterministic fault in a controlled way in many different configurations that trigger the failure. During the inspection of the faults, they could face that large part of the test input data that is completely irrelevant to understand the fault. MRDebug automatically obtains a minimal part of the test input data that triggers the failure, helping the tester/developer to simplify this inspection to a minimal part. In practice, MRDebug can both enhance the debugging of the *MapReduce* applications and decrease the time to fix the design faults.

As future work we plan to extend MRDebug implementation to support other *MapReduce* frameworks like *Spark* or *Flink*. Another future research line is to repair/fix [88] the programs automatically during runtime using a self-adapt technique based on PDCA methodology. Once a program is executed in production, the testing technique MRTTest will detect the design faults. Next, MRDebug will locate the root cause of the fault. Finally, the program should be automatically repaired to pass the tests and continue the execution.

REFERENCES

- [1] *Information Technology—Big Data—Overview and Vocabulary*, ISO IEC 20546:2019, 2019.
- [2] “Apache Hadoop: Open-source software for reliable, scalable, distributed computing.” Accessed: Jan. 23, 2017. [Online]. Available: <https://hadoop.apache.org/>
- [3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, p. 10.
- [4] “Apache Flink: Scalable batch and stream data processing.” Accessed: Feb. 20, 2017. [Online]. Available: <https://flink.apache.org>
- [5] M. Kim, T. Zimmermann, R. Deline, and A. Begel, “Data scientists in software teams: State of the art and challenges,” *IEEE Trans. Softw. Eng.*, vol. 44, no. 11, pp. 1024–1038, Nov. 2018.
- [6] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. Symp. Oper. Syst. Des. Implement. (OSDI)*, 2004, pp. 137–149.
- [7] M. C. Schatz, “CloudBurst: Highly sensitive read mapping with MapReduce,” *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, Jun. 2009.
- [8] H. Kocakulak and T. T. Temizel, “A Hadoop solution for ballistic image analysis and recognition,” in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2011, pp. 836–842.
- [9] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, “An analysis of traces from a production MapReduce cluster,” in *Proc. 10th IEEE/ACM Int. Conf. Cluster, Cloud Grid Comput.*, 2010, pp. 94–103.
- [10] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, “Hadoop’s adolescence,” in *Proc. VLDB Endowment*, 2013, vol. 6, no. 10, pp. 853–864.
- [11] T. Xiao et al., “Nondeterminism in MapReduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs,” in *Proc. 36th Int. Conf. Softw. Eng. (ICSE Companion)*, 2014, pp. 44–53.
- [12] J. Moran, C. de la Riva, and J. Tuya, “MRTree: Functional testing based on MapReduce’s execution behaviour,” in *Proc. Int. Conf. Future Internet Things Cloud*, 2014, pp. 379–384.
- [13] L. C. Camargo and S. R. Vergilio, “Classificação de defeitos para programas MapReduce: Resultados de um estudo Empírico,” in *Proc. 7th Brazilian Workshop Syst. Automat. Softw. Test.*, Brasília: Congresso Brasileiro de Software: Teoria e Prática (CBSOFT), 2013.
- [14] J. Moran, A. Bertolino, C. de la Riva, and J. Tuya, “Automatic testing of design faults in MapReduce applications,” *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 717–732, Sep. 2018.
- [15] H. Zhou, J. G. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin, “An empirical study on quality issues of production Big Data platform,” *Proc. Int. Conf. Softw. Eng.*, vol. 2, Aug. 2015, pp. 17–26.
- [16] M. Bagherzadeh and R. Khatchadourian, “Going big: A large-scale study on what big data developers ask,” in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf./Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2019, pp. 432–442.
- [17] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, 2011, pp. 199–209.
- [18] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.
- [19] A. Zeller, “Yesterday, my program worked. Today, it does not. Why?” *ACM SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 253–267, 1999.
- [20] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.
- [21] D. Lewis, “Causation,” *J. Philos.*, vol. 70, no. 17, pp. 556–567, 1973.
- [22] Z. Xu, M. Hirzel, and G. Rothermel, “Semantic characterization of MapReduce workloads,” in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2013, pp. 87–97.
- [23] J. M. Glenford, *The Art of Software Testing*, New York, NY, USA: Wiley, 1979.
- [24] M. Grindal, J. Offutt, and S. F. Andler, “Combination testing strategies: A survey,” *Softw. Test. Verification Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.
- [25] C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Comput. Surv.*, vol. 43, no. 2, pp. 1–29, 2011.
- [26] T. Chen, S. Cheung, and S. Yiu, “Metamorphic testing: A new approach for generating next test cases,” *Dept. Comput. Sci. Hong Kong Univ. Sci. Technol., Hong Kong, Tech. Rep. HKUST-CS98-01*, 1998, pp. 1–11.
- [27] S. Segura, G. Fraser, A. B. Sanchez and A. Ruiz-Cortes, “A survey on metamorphic testing,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 805–824, Sep. 2016.
- [28] E. J. Weyuker, “On testing non-testable programs,” *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.
- [29] T. Reps, T. Ball, M. Das and J. Larus, “The use of program profiling for software maintenance with applications to the year 2000 problem,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 22, no. 6, pp. 432–449, 1997.
- [30] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, “Empirical investigation of the relationship between spectra differences and regression faults,” *Softw. Test. Verif. Rel.*, vol. 10, no. 3, pp. 171–194, 2000.
- [31] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, “No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist,” Research Note, Univ. Coll. London, 2014.
- [32] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, “Spectrum-based fault localization in software product lines,” *Inf. Softw. Technol.*, vol. 100, pp. 18–31, 2018.
- [33] S. Babu, “Towards automatic optimization of MapReduce programs,” in *Proc. 1st ACM Symp. Cloud Comput. (SoCC)*, 2010, pp. 137–142.
- [34] “JobConf.” Apache Hadoop. Accessed: Dec. 2, 2021. [Online]. Available: <https://hadoop.apache.org/docs/r3.3.1/api/org/apache/hadoop/mapred/JobConf.html>
- [35] A. Groce, S. Chaki, D. Kroening, and O. Strichman, “Error explanation with distance metrics,” *Int. J. Softw. Tools Technol. Trans.*, vol. 8, no. 3, pp. 229–247, 2006.
- [36] H. A. De Souza, M. L. Chaim, and F. Kon, “Spectrum-based software fault localization: A survey of techniques, advances, and challenges,” 2016, *arXiv:1607.04347*.
- [37] L. Naish, H. J. Lee, and K. Ramamohanarao, “A model for spectra-based software diagnosis,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–32, 2011.
- [38] M. Patton, *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*, Newbury Park, CA, USA: Sage, 2014.
- [39] S. Baltes and P. Ralph, “Sampling in software engineering research: A critical review and guidelines,” *Empir. Softw. Eng.*, vol. 27, no. 4, Jul. 2022, Art. no. 94.
- [40] M. Interlandi et al., “Titian: Data provenance support in Spark,” in *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 216–227, 2016.

- [41] M. A. Gulzar, V. Tech, and M. Kim, "OptDebug: Fault-inducing operation isolation for dataflow applications," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 359–372.
- [42] F. Grigorjev, N. Lascano, and J. L. Staude, "A fault seeding experience," in *Proc. Simposio Argentino de Ingenieria de Softw. (ASSE)*, pp. 1–14.
- [43] "Data mining and machine learning based on MapReduce." Open Ankus. Accessed: Feb. 13, 2024. [Online]. Available: <http://www.openankus.org/>
- [44] B. Rivas, J. Merino, M. Serrano, I. Caballero, and M. Piattini, "I8KIDQ-BigData: I8K architecture extension for data quality in big data," in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9382, 2015, pp. 164–172.
- [45] "Movies analysis implemented in MapReduce." GitHub. Accessed: Feb. 13, 2024. [Online]. Available: https://github.com/adityaundirwadkar/mapreduce-programming/tree/master/example_1
- [46] "Treelogic S.L." Treelogic. Accessed: Feb. 13, 2024. [Online]. Available: www.treelogic.com
- [47] "PageRank implemented in MapReduce." GitHub. Accessed: Feb. 13, 2024. [Online]. Available: <https://github.com/JohandeGraaf/PageRank>
- [48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, Berlin, Germany: Springer Science & Business Media, 2012.
- [49] J. Morán, A. Bertolino, C. de la Riva, and J. Tuya, "Supplemental material for: automatic debugging of design faults in MapReduce applications." Zenodo. [Online]. Available: <https://doi.org/10.5281/zenodo.7778710>
- [50] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 21, no. 6, pp. 803–827, Apr. 2012.
- [51] H. Coolican, *Research Methods and Statistics in Psychology*. London, U.K.: Hodder, 2009.
- [52] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 4, pp. 573–597, 2009.
- [53] B. Jiang, Z. Zhang, W. K. Chan, T. H. Tse, and T. Y. Chen, "How well does test case prioritization integrate with statistical fault localization?" *Inf. Softw. Technol.*, vol. 54, no. 7, pp. 739–758, 2012.
- [54] J. Cohen, *Statistical Power for the Behaviour Sciences*, Cambridge, MA, USA: Academic Press, 1977.
- [55] R. Abreu, P. Zoetevej, and A. J. C. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Testing: Academic Ind. Conf. Pract. Res. Tech. (TAIC PART-Mutation)*, 2007, pp. 89–98.
- [56] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006.
- [57] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs," in *Proc. 2019 27th ACM Joint Meeting Eur. Softw. Eng. Conf./Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2019, pp. 556–566.
- [58] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf./Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2021, pp. 881–892.
- [59] T. D. Cook, D. T. Donald, and T. Campbell, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Boston, MA, USA: Houghton Mifflin, 1979.
- [60] R. Malhotra, *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*.
- [61] J. Morán, C. De La Riva, and J. Tuya, "Testing MapReduce programs: A systematic mapping study," *J. Softw. Evol. Process.*, vol. 31, no. 3, 2019, pp. 1–29.
- [62] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 402–411.
- [63] V. Jagannath, Z. Yin, and M. Budiu, "Monitoring and debugging DryadLINQ applications with daphne," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2011, pp. 1266–1273.
- [64] M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim, "Automated debugging in data-intensive scalable computing," in *Proc. Symp. Cloud Comput. (SoCC)*, 2017, pp. 520–534.
- [65] A. Chapman, P. Missier, G. Simonelli, and R. Torlone, "Capturing and querying fine-grained provenance of preprocessing pipelines in data science," in *Proc. VLDB Endowment*, 2020, vol. 14, no. 4, pp. 507–520.
- [66] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker, "Interactions with big data analytics," *Interactions*, vol. 19, no. 3, pp. 50–59, 2012.
- [67] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesh: BlackBox diagnosis of MapReduce systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 3, pp. 8–13, 2010.
- [68] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Mochi: Visual log-analysis based tools for debugging hadoop," in *Proc. Conf. Hot Topics Cloud Comput.*, 2009, pp. 1–5.
- [69] E. Garduno, S. P. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "Theia: visual signatures for problem diagnosis in large hadoop clusters," in *Proc. Int. Conf. Large Installation Syst. Admin.*, vol. 2, 2012, pp. 33–42.
- [70] N. Khousainova, M. Balazinska, and D. Suciu, "PerfXplain: Debugging MapReduce job performance," in *Proc. VLDB Endowment*, Mar. 2012, vol. 5, no. 7, pp. 598–609.
- [71] C. Olston and B. Reed, "Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 1221–1224.
- [72] M. Geceer, O. Nierstrasz, and H. Osman, "Debugging spark applications: A study on debugging techniques of Spark developers," Master thesis, Univ. Bern, Bern, 2020.
- [73] H. Feng, J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn, "A method-level test generation framework for debugging Big Data applications," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, pp. 221–230, Jan. 2019.
- [74] R. Diestelkämper, S. Lee, B. Glavic, and M. Herschel, "Debugging missing answers for spark queries over nested data with breadcrumb," in *Proc. VLDB Endowment*, Jul. 2021, vol. 14, no. 12, pp. 2731–2734.
- [75] Z. Wang, H. Zhang, T. H. P. Chen, and S. Wang, "Would you like a quick peek? Providing logging support to monitor data processing in big data applications," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf./Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2021, pp. 516–526.
- [76] J. Moran, A. Bertolino, C. de la Riva, and J. Tuya, "Towards ex vivo testing of MapReduce applications," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, 2017, pp. 73–80.
- [77] A. Kumar, Z. Wang, S. Ni, and C. Li, "Amber: A debuggable dataflow system based on the actor model," in *Proc. VLDB Endowment*, Jan. 2020, vol. 13, no. 5, pp. 740–753.
- [78] Z. Wang, "Understanding the challenges and assisting developers with developing Spark applications," in *Proc. Int. Conf. Softw. Eng.*, May 2021, pp. 132–134.
- [79] M. A. Gulzar et al., "BigDebug: Debugging primitives for interactive big data processing in Spark," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 784–795.
- [80] M. Marra, G. Polito, and E. Gonzalez Boix, "A debugging approach for live Big Data applications," *Sci. Comput. Program*, vol. 194, Aug. 2020, Art. no. 102460.
- [81] B. Contreras-Rojas, J. A. Quiané-Ruiz, Z. Kaoudi, and S. Thirumuruganathan, "TagSniff: Simplified big data debugging for dataflow jobs," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, Nov. 2019, pp. 453–464.
- [82] A. Dave, M. Zaharia, S. Shenker, and I. Stoica, "Arthur: Rich post-facto debugging for production analytics applications," Citeseer, Tech. Rep. 2013.
- [83] D. Logothetis, S. De, and K. Yocum, "Scalable lineage capture for debugging DISC analytics," in *Proc. 4th Annual Symp. Cloud Comput. (SOCC)*, 2013, pp. 1–15.
- [84] E. Bergen and S. Edlich, "Post-debugging in large scale big data analytic systems," in *Datenbanksysteme Für Bus., Technologie Und Web*, 2017, pp. 65–74.
- [85] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs! Understanding and dealing with over-designed configuration in system software," in *Proc. 2015 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 307–319.
- [86] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An evolutionary study of configuration design and implementation in cloud systems," in *Proc. 2021 IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 188–200.
- [87] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," *Proc. 28th ACM Jt. Meet. Eur. Softw. Eng. Conf./Symp. Found. Softw. Eng. (ESEC/FSE)*, Nov. 2020, pp. 362–374.
- [88] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, Jan. 2019.



Jesús Morán received the Ph.D. degree in computing from the University of Oviedo. He is an Assistant Professor with the University of Oviedo. He is a Member of the Software Engineering Research Group (GIIS, giis.uniovi.es). His research interests include software testing, Big Data technologies, and distributed programming.



Claudio de la Riva received the Ph.D. degree in computing from the University of Oviedo. He is an Associate Professor with the University of Oviedo. He is a Member of the Software Engineering Research Group (GIIS, giis.uniovi.es). His research interests include software verification and validation and software testing, mainly focused on testing database applications and services.



Antonia Bertolino received the M.S. degree in electronic engineering from the University of Pisa. She is a Research Director with the ISTI-CNR (the Institute of Information Science and Technologies “A. Faedo” of the Italian National Research Council), Pisa, Italy. Her research interests include software testing. She is an Associate Editor of Wiley *Journal of Software Evolution and Process*, and serves as the Software Testing Area Editor of the Elsevier *Journal of Systems and Software*. She has been the General Chair of the 2015 International Conference on Software Engineering held in Florence (Italy).



Javier Tuya (Member, IEEE) received the Ph.D. degree in engineering from the University of Oviedo, Oviedo, Spain, in 1995. He is a Professor with the University of Oviedo, Spain, where he is the Research Leader of the Software Engineering Research Group. He is the Director of the Indra-Uniovi Chair and worked in the development of the new ISO/IEC/IEEE 29119 Software Testing Standard as Member of the ISO WG26 Working Group and as Convener of the corresponding UNE National Body Working Group. His research interests include software engineering software testing for database applications and system testing. He is a member of the IEEE Computer Society and ACM.