Poster: Efficiently Finding Minimal Failing Input in MapReduce Programs

Muhammad Sohaib Ayub, Junaid Haroon Siddiqui
School of Science and Engineering, Lahore University of Management Sciences, Lahore, Pakistan
{15030039,junaid.siddiqui}@lums.edu.pk

ABSTRACT

Debugging of distributed computing model programs like MapReduce is a difficult task. That's why prior studies only focus on finding and fixing bugs in early stages of program development. Delta debugging tries to find minimal failing input in sequential programs by dividing inputs into subsets and testing these subsets one-by-one. But no prior work tries to find minimal failing input in distributed programs like MapReduce. In this paper, we present MapRedDD, a framework to efficiently find minimal failing input in MapReduce programs. MapRedDD employs failing input selection technique, focused on identifying the failing input subset in the single run of MapReduce program with multiple input subsets instead of testing each subset separately. This helps to reduce the number of executions of MapReduce program for each input subset and overcome the overhead of job submission, job scheduling and final outcome retrieval. Our work can efficiently find the minimal failing input in the number of executions equal to the number of inputs to MapReduce program N as opposed to the number of executions of MapReduce program equal to the number of input subsets $2^N - 1$ in worst case for binary search invariant algorithm to find minimal failing input.

CCS CONCEPTS

• **Software and its engineering** → Software testing and debugging;

KEYWORDS

Software Verification, Delta Debugging, MapReduce

ACM Reference Format:
Muhammad Sohaib Ayub, Junaid Haroon Siddiqui. 2018. Poster: Efficiently Finding Minimal Failing Input in MapReduce Programs. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195084

1 INTRODUCTION AND BACKGROUND

MapReduce[4] is a simple data-programming model designed for processing of large data sets in the distributed manner using a large number of commodity machines or nodes. MapReduce programs are automatically parallelized, highly scalable, fault tolerant, I/O scheduled and most of the real world tasks can be solved using MapReduce. The runtime system of MapReduce takes care of partitioning the input data, scheduling execution of the program on different nodes, fault tolerance and intra-node communication. This allows programmers without any experience of distributed and parallel systems to easily utilize the resources of a large distributed system.

Delta debugging[6] tries to systematically simplify the input that leads the program to failure. It iteratively reduces the size of input to smallest input that causes failure for easier debugging of programs. It simplifies failure-inducing circumstances using a variation of binary search to remove individual components of a failing test case such that further removing any element will cause failure to disappear. Minimizing delta debugging algorithm takes failing test case due to the set of inputs or circumstances. It reduces the set of inputs by successively testing the program using the input subset obtained from the variation of the binary search algorithm i.e. iteratively dividing the set of input into subsets. The algorithm stops when a minimal failing set of inputs is reached and removing any single input will cause failure to disappear.

In case of MapReduce programs, large number of inputs can cause MapReduce program to fail. Finding the minimal failing input in MapReduce programs will help the developers to easily debug it. We have proposed a framework, called MapRedDD, for efficiently finding minimal failing input extending delta debugging algorithm. MapRedDD executes multiple inputs subsets in single program execution and then finds the failing subset to reduce failure circumstances. Each execution of MapReduce program requires submitting the job, uploading inputs to the cluster, executing MapReduce jobs and downloading the results from the cluster which takes a lot of time and computation resources for a single execution of MapReduce program. Therefore, MapRedDD reduces the number of MapReduce executions for finding minimal failing input.

2 TECHNIQUE AND IMPLEMENTATION

MapRedDD is a framework to efficiently find minimal failing input in MapReduce programs. MapReduce program is tested with the set of inputs which might result Failed, Passed or Unresolved test case as shown in Figure 1. On the basis of the result of test case execution, the MapRedDD algorithm reduces the set of inputs till it finds the set of failing inputs from which removing any input will cause failure to disappear.

For example, we have a program that counts the number of integers in the set of inputs and fails if any of the inputs is not an integer. When we run delta debugging algorithm on failing the set of input as shown in figure 2(a), the MapReduce program fails due

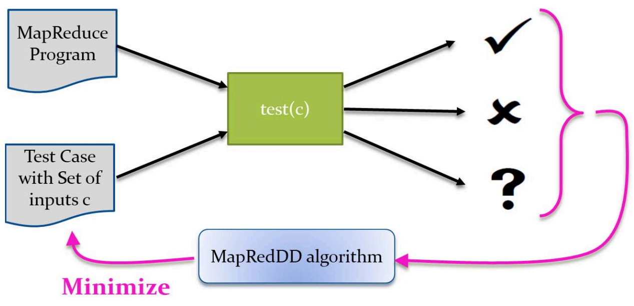
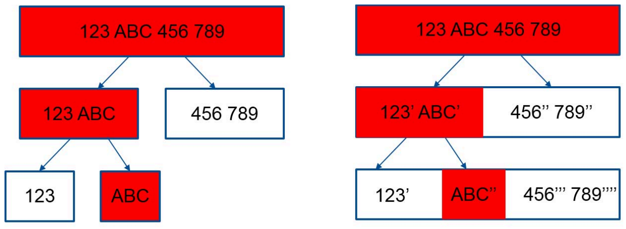


Figure 1: Proposed Framework (MapRedDD)

to the presence of string "ABC" in the set of inputs. Delta debugging algorithm divides the set of inputs into two subset sets, runs MapReduce program on each of subset and tries to find the failing subset of input. After finding the failing subset, delta debugging again iteratively splits and checks the failing set of inputs into two subsets until the minimal failing input is found, which is "ABC" in this example. As we can see, the number of MapReduce executions for finding the failing input is same as the number of input subsets for N inputs i.e. $2^N - 1$ in worst case which is very expensive in terms of resources as well as time.

Our proposed MapRedDD algorithm finds the minimal failing input in an efficient way. It splits the input set into subsets in the same way as delta debugging does. After that instead of testing each of the subsets in separate MapReduce execution, MapRedDD passes all the subsets in the single execution of MapReduce program as shown in Figure 2(b). For this purpose, MapRedDD taint keys by adding subset identification to the actual keys. This information is used to identify the failing input subset after the execution of MapReduce program. To handle these tainted keys by the MapReduce program, MapRedDriver class is implemented which extracts the subset information before sending these tainted keys and hence original keys are sent to MapReduce program. In case of any failure, MapReduce driver can report the failing subset information. After the execution of map function, the subset information is again added to the map function output so that no two keys from different subsets should be sent to the same reducer and does not depict that these are same keys. After receiving the tainted keys in the reduce function, MapRedDD again extracts the subset information from the keys same as it was done in map function. For any failure in reduce function, MapRedDD can output the failure causing subset information. After the reduce function, these keys are again added to the output of the reduce function which is the output of the program with tainted keys. If MapRedDD is unable to find any failure in the complete execution of MapReduce program i.e. in all the input subsets of the program under test then it is considered as unresolved. After getting the information about the passed, failed and unresolved subsets using tainting keys, MapRedDD uses the delta debugging algorithm to further split the input into the subsets and find the minimal failing input.

Let's consider the previous integer count example again for our MapRedDD algorithm. The program fails in the shown input due to the presences of string "ABC" in the input set. MapRedDD divides the input into two subsets and taint the inputs of both subsets so that after running both subsets in the same execution of MapReduce program, it can be detected that which subset caused the program



(a) Delta Debugging (b) MapRedDD Algorithm

Figure 2: Technique Visualization

to fail. Both of these tainted input subsets then passed into the MapRedDD framework which runs both of these subsets in same MapReduce execution. By doing this MapRedDD efficiently reduces the number of MapReduce executions from number of subsets $2^N - 1$ in worst case to the only number of inputs N. After the execution of both these subsets, there might be failing, passing or unresolved results. MapRedDD uses this information to further split the input into subsets till the minimal failing input is found from which removing any input will cause failure to disappear. MapRedDD algorithm actually tries to find failure-inducing input using an efficient algorithm which finds the failing input subset by running multiple tainted input subsets in a single run of MapReduce program.

3 RELATED WORK

This paper tries to reduce failure causes in MapReduce[4] programs using previous work on delta debugging[6]. Finding minimal failing input in MapReduce programs using original delta debugging algorithm is very expensive in terms of time and resource usage which is efficiently solved using our proposed MapRedDD. There is not a single work which tries to find the failure causes for the MapReduce programs and all the previous efforts tries to test the MapReduce programs either using mocking or unit testing.

Mockito[2] and PowerMock[3] are the famous mocking frameworks used to create mock objects and verify the system behavior. For example, Map and Reduce function write their output to context object and mocking context object is used to verify whether the map and reduce function are writing the output correctly.

Sometimes, mocking may not be sufficient and unit testing can offer an additional level of coverage. Since the early days of MapReduce programs, people have been trying to unit test MapReduce programs but it remains the challenging task because Mapper and Reducers run in distributed environment across many JVMs in the cluster of machines[5]. MRUnit[1] is a Java library to unit test MapReduce Jobs. It works in isolation and does not require MapReduce demons to be running because distributed nature of MapReduce programs add extra complexity. But this isolation also causes the lack of interaction with cluster and distributed file system.

REFERENCES

[1] 2018. Apache MRUnit. https://mrunit.apache.org/. (2018).
[2] 2018. Mockito. https://code.google.com/p/mockito/. (2018).
[3] 2018. PowerMock. https://code.google.com/p/powermock/. (2018).
[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[5] Tom White. 2012. *Hadoop: The Definitive Guide.* " O'Reilly Media, Inc.".
[6] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.