New Ideas Track: Testing MapReduce-Style Programs

Christoph Csallner
csallner@uta.edu

Leonidas Fegaras
fegaras@uta.edu

Chengkai Li
cli@uta.edu

Computer Science and Engineering Department
University of Texas at Arlington
Arlington, TX 76019, USA

ABSTRACT

MapReduce has become a common programming model for processing very large amounts of data, which is needed in a spectrum of modern computing applications. Today several MapReduce implementations and execution systems exist and many MapReduce programs are being developed and deployed in practice. However, developing MapReduce programs is not always an easy task. The programming model makes programs prone to several MapReduce-specific bugs. That is, to produce deterministic results, a MapReduce program needs to satisfy certain high-level correctness conditions. A violating program may yield different output values on the same input data, based on low-level infrastructure events such as network latency, scheduling decisions, etc. Current MapReduce systems and tools are lacking in support for checking these conditions and reporting violations.

This paper presents a novel technique that systematically searches for such bugs in MapReduce applications and generates corresponding test cases. The technique works by encoding the high-level MapReduce correctness conditions as symbolic program constraints and checking them for the program under test. To the best of our knowledge, this is the first approach to addressing this problem of MapReduce-style programming.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, reliability*

General Terms

Algorithms, Reliability, Verification

Keywords

MapReduce, dynamic symbolic execution, test generation

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

1. INTRODUCTION

MapReduce-style programming is a new paradigm in large-scale data processing that has attracted a lot of attention, both in industry and academia. In the six years since its publication, according to Google Scholar, the original MapReduce paper [3] has been cited over 2,000 times. In practice, several large organizations have implemented the MapReduce paradigm, e.g., in Apache/Yahoo! Hadoop [14] and Pig [8], Apache/Facebook Hive [13], Google Sawzall [9], and Microsoft Dryad [5].

MapReduce programs are especially well-suited for processing very large amounts of data (e.g., two petabyte worth of log files) on a large number of commodity machines (e.g., a 10,000 node Linux cluster). However, MapReduce systems are also increasingly being used for other tasks that were traditionally the domain of parallel database management systems, for example, data warehousing [13, 11].

There are many notable examples of MapReduce usages in a variety of application domains. Amazon Elastic Compute Cloud (EC2) provides cloud computing services on Hadoop. Yahoo! used Hadoop to sort a terabyte of data in just 209 seconds, which is the fastest with regard to the Jim Gray terabyte sorting benchmark. Facebook uses Hive as a data warehouse solution for internal reporting and ad-hoc data analysis [13].

Programming in MapReduce is complicated by two main factors. (1) To produce deterministic results, a MapReduce system requires user programs to satisfy certain high-level correctness conditions. That is, if a MapReduce program violates these correctness conditions, the output values emitted by that program may vary based on low-level infrastructure events such as network latency, scheduling decisions, etc. This means that running the same MapReduce program again on the same input data may yield different results. For many tasks, such non-determinism in their program behavior is clearly a bug programmers would like to avoid. However, (2), neither the MapReduce execution system nor current tools we are aware of check these conditions or warn the user when they may be violated.

This paper addresses this problem with a novel technique that systematically searches for such bugs in MapReduce programs and generates test cases that allow programmers to reproduce the bug. Our technique works by encoding the high-level MapReduce correctness conditions as symbolic program constraints and checking them for the particular user program under test. We implement the technique in a dynamic symbolic execution [4, 1] framework, which allows us to generate test cases that exhibit the bugs found

by our technique. To the best of our knowledge, this is the first approach to addressing this problem of MapReduce-style programming.

This paper is phrased in the terms of the Java programming language, which is the language used in Hadoop, the most popular open-source implementation of MapReduce. However, the presented technique is independent of a particular language or system. That is, one can encode the program constraints and MapReduce correctness conditions for programs written in different languages that target different MapReduce implementations. So the same high-level technique can be applied to MapReduce programs running on one of the common MapReduce systems including Hadoop and the original Google MapReduce system. The technique can equally be adapted to other mainstream programming languages such as C++ and C#.

1.1 Novelty and Originality of our Approach

To the best of our knowledge, this is the first approach that uses dynamic symbolic execution to automatically generate test cases that can reveal certain types of bugs in general MapReduce programs. Instead of testing these programs on the entire data sets, which are often very large, our method allows programmers to discover hard-to-find bugs using very small synthetic data sets. Dynamic symbolic execution is precise and works well for relatively small programs, which makes it attractive for map/reduce programs. Although our paper is focused on enforcing the commutativity property of the reduce function, it has the potential of becoming a general framework for automated MapReduce testing. There is no other published paper on a related approach by the authors or others. We expect that feedback from the reviewers and the workshop participants will help us identify more cases for program testing in a MapReduce environment.

2. CORRECTNESS CONDITONS

A MapReduce system requires that a user program is expressed as three separate user-defined functions, which are named map, reduce, and combine.

Map: A MapReduce job may invoke a given map function on many nodes in parallel, each processing a portion of the input data. Each invocation produces a list of (key, value) pairs. The set of all (key, value) pairs produced by this job will be passed as input to a reduce function, which may also be running on multiple nodes.

Reduce: A reduce function receives as input a key and a list of values that are associated with that key. The value list is ordered according to the order in which the values are received on the reducer node, which can be influenced by network latency, etc. To produce deterministic results, the implementation of the user-defined reduce function must be independent of the input value order. We call this requirement REQ-1.

More formally, for each input list L, the reduce function has to return the same result for each permutation P of that list:

$$\forall L, P : reduce(k, L) = reduce(k, P(L)) \qquad \text{(REQ-1)}$$

Combine: A combine function is used in an optional optimization step that combines values by key on a map node. This is useful as reducing values locally, before transmitting them to the reducer, can save network bandwidth. MapReduce systems may freely decide if and how often to apply a combine function. Like REQ-1, to produce deterministic results, the number of times the system invokes a user-defined combine function must not influence the values produced by the downstream reduce functions (REQ-2). In this paper, for space reasons, we focus on checking REQ-1.

3. EXAMPLE: REDUCE BUG

Consider the example reduce method in Listing 1. The programmer intends to report the average of the top 100 salaries for each unique key, if that average is larger than 100,000. A key corresponds to, say, a department. However, the code does not satisfy the MapReduce requirement that reduce must be independent of the order in which it receives the input values (REQ-1). That is, to compute the average, instead of finding the highest 100 salaries in the list, the code just uses the first 100 values. So the value returned by the reduce function is determined by the salaries that happen to be at the head of the input list.

```
1 public void reduce(String key, Iterator<Integer> salaries) {
    int sum = 0; int i = 0;
3   while (salaries.hasNext() && i<100) {
      sum += salaries.next();
5     i += 1;
    }
7   emit((i>0 && sum/i > 100000)? sum/i : −1);
  }
```

Listing 1: The average of the top 100 salaries for each unique key, if that average is larger than 100,000.

This bug may be due to a number of reasons. One, the programmer may not be aware of correctness condition REQ-1. Second, the programmer may mistakenly assume that the input values are already ordered when they reach the reduce function. This misunderstanding may stem from the fact that MapReduce systems, such as Hadoop, have various built-in value sorting techniques, but do not always apply them. The misunderstanding may also stem from the programmer's knowledge of the order in which salaries are being stored in the original input files. Such order is to be destroyed by the parallel processing of MapReduce.

To find bugs such as the one in Listing 1, at a high level, we need to check if there exists an input list L and a permutation P, such that $reduce(k, L) \neq reduce(k, P(L))$. However, answering that question is hard, as the number of candidate lists and their permutations is not finite. E.g., for Listing 1, the bug is only exposed for a test case that supplies a certain kind of input list, one that has more than 100 elements and the average value of the first 100 elements is larger than 100,000. In addition, the test case needs to contain a certain kind of permutation of that list, one that replaces an element at position less than 100 with one at position greater than 100. Hence, trying a lot of randomly generated test cases can be wasteful. On the other hand, finding a desired test case is hard and requires deep analysis of the program and the MapReduce correctness conditions.

Although the question is hard in general, we observe that the number of distinct execution paths through a reduce method is typically small, since the main purpose of a reduce method is to aggregate the values it receives. We therefore do not expect much complex branching decisions based on

the input list elements, as element filtering is typically done in the map method. Once data arrives at the reduce method, it is already filtered and merely needs to be combined. This means that a white-box code analysis system has a chance to explore a meaningful portion of the execution paths that are feasible in a reduce method. In the example of Listing 1, if we fix the input list length to, say, 128, the reduce method can be covered with just two distinct execution paths, which differ in whether sum/i is greater than 100,000. Dynamic symbolic execution is a technique that is well-suited to derive such a small set of execution paths precisely.

Using dynamic symbolic execution, we can derive the path condition of the execution paths from executing the reduce method and tracking the branches taken. For example, we get one path condition conjunction of: $salaries \neq null$, $salaries.length > 99$, $(salaries[0] + .. + salaries[99])/100 > 100000$. By tracking all values symbolically, we also get the symbolic expression of any values returned, in this case the value $((salaries[0] + .. + salaries[99])/100)$. At this point we can encode the first MapReduce correctness condition (REQ-1) directly on these symbolic values and use an automated constraint solver to check if they are satisfiable. In order to do that, we encode the permutation of the input list via symbolic integer variables, which encode the indices of the permuted list. That is, we create 128 variables $a_0, .., a_{127}$, require that they take on distinct values, and encode the correctness condition as the path condition conjoined with $(salaries[0] + .. + salaries[99])/100 \neq (salaries[a_0] + .. + salaries[a_{99}])/100$. If there is a solution, and this constraint system may have several, we can retrieve both the list elements and the permutation from it and compile them into a test case. The test case invokes the reduce function once on the solution list and once on the permuted solution list, compares the results, and issues a warning if the results are different.

4. PROPOSED APPROACH

Our technique works by exploring the MapReduce program under test with dynamic symbolic execution. It (1) derives symbolic expressions for path conditions and return values, (2) maintains them in a novel indexed execution tree (Section 4.1), (3) encodes with those symbolic expressions the MapReduce correctness conditions of Section 2, (4) uses an off-the-shelf constraint solver to infer program input values that will violate the correctness conditions, and finally (5) converts the violating input values to test cases. All the steps can be automated, yielding a fully automated test case generator for MapReduce programs.

Listing 2 contains the main loop of this process. It starts by creating default values for the input parameters of the reduce function in a MapReduce program, e.g., a key value of null and a list of two default values such as (0, 0). On the symbolic side, we create a corresponding key variable and a symbolic array variable that will represent the input list. Then we evaluate the reduce function, both concretely on the initial values and symbolically on the variables, where the symbolic execution merely traces the single path taken by the concrete execution. Using the results of the symbolic execution, we can now check, in the check_path method, if the correctness condition REQ-1 can be violated on the same execution path. That is, in check_req1 (Listing 3), we assert the path condition of the executed path symbolically, together with the condition that values returned by the reduce function are different for an input list and a permutation of that list.

```
   /* The reduce method has two parameters: key and values */
 2 test_reduce(meth<key_type, iterator<value_type>> reduce) {
     key := get_default_value(key_type);
 4   Key := get_fresh_var(key_type);
     value_cnt := 2; // length of input list
 6   values := get_default_array(value_type, value_cnt);
     Values := Get_fresh_array_var(value_type);
 8   (Path, Result) := Eval(reduce, key, values, Key, Values);
     check_path(Key, Values, value_cnt, Path, Result);
10   while (unexplored_paths(value_cnt)) {
       PathCond := next_unexplored_path(value_cnt); // path p
12     Assert: PathCond; // check if path p is feasible
       Assert: Length(Values) == Value_cnt;
14     M := GetModel(); // constraint model/solution
       if (is_empty(M))
16       continue; // no input that would trigger path p
       (key, values) := M(Key, Values); // new input from model
18     (Path, Result) := Eval(reduce, key, values, Key, Values);
       check_path(Key, Values, value_cnt, Path, Result);
20   }
     value_cnt := next_value();
22 }

24 check_path(Var Key, Var Values, int cnt, Cond Path, Expr Res)
   {
26   add_to_execution_tree(cnt, Path, Res);
     check_req1(Key, Values, cnt, Path, Res, Path, Res);
28   foreach (SibP, SibRes) in exec_tree_paths(cnt)
       check_req1(Key, Values, cnt, Path, Res, SibP, SibRes);
30 }
```

Listing 2: Main dynamic symbolic test case generator loop. Symbolic entities are capitalized. Eval evaluates the given program both concretely and symbolically; Assert adds a conjunct to the constraint system; GetModel solves the current constraint system, retrieves a solution, and resets the constraint system; *SibP* and *SibRes* are a sibling path and its respective return value; being a sibling means using an input list of the same length.

4.1 Indexed Execution Tree

A dynamic symbolic execution system maintains a tree of the execution paths that it already explored. This execution tree is a binary tree, in which each node represents the outcome of a branch condition, expressed in terms of the symbolic program input variables. For the example of Listing 1, one tree node is $salaries.length > 3$, which has the child node $salaries.length > 4$, etc. To discover additional execution paths, the system examines each single parent node, that is, each node that has exactly one child node. Inverting the branch decision outcome of such a single child node yields a candidate leaf node of an unexplored execution path, which is implemented in next_unexplored_path. The test_reduce algorithm uses this technique to automatically check as many execution paths as possible.

In our system, the execution tree is extended in two ways. (1) Each leaf node stores the values (typically, one value) returned during the given execution. (2) We index the leaf node by the length of the input list that triggered that execution. This is useful, as it allows us to check if a path triggered by an input list of length l may return a value that is different than the value returned on a *sibling path*. A sibling path is any other path that is triggered by an input list of the same length l.

```
check_req1(Expr Key, Var Values, int cnt, Cond PathCond,
           Expr Res, Cond SibPath, Expr SibRes) {
  p := PrepareIndexVariables(cnt);
  Assert: PathCond;
  Assert: SubstituteIndices(SibPath, p);
  Assert: Res != SubstituteIndices(SibRes, p);
  M := GetModel(); // constraint model/solution
  if (has_solution(M)) {
    log: "permutations that yield different results:"
    M(Key), M(Values[0]), .., M(Values[cnt−1]) "vs."
    M(Key), M(Values[M(p[0])]), .., M(Values[M(p[cnt−1])]);
  }
}

IntVar[] PrepareIndexVariables(int cnt) {
  p := new IntVar[cnt]; // position indices
  foreach i in [0,cnt−1] {
    p[i] := Get_fresh_var(int);
    Assert: 0 ≤ p[i] < cnt;
  }
  Assert: Distinct(p[0], .., p[cnt−1]);
  return p;
}
```

Listing 3: Checking REQ-1. Symbolic entities are capitalized. PrepareIndexVariables returns a list of symbolic integer variables p_0, .., p_{cnt-1}; SubstituteIndices replaces in the given expression each symbolic literal index i with symbolic variable p_i.

4.2 Input Length Heuristic

At a high level, the test_reduce algorithm tests if the reduce method satisfies REQ-1 for a number of paths the reduce execution can take. Each path is triggered by an input list of a certain length. Clearly, in testing, we cannot check a method for all possible inputs. In the test_reduce algorithm, we can address this problem by using a heuristic for picking representative input lengths. That is, test_reduce starts with input list of length 2, as shorter lists cannot expose a violation of REQ-1. For subsequent iterations, the heuristic is implemented by the next_value function. We currently experiment with a binary back-off scheme, in which each iteration doubles the length of the input list. Using this heuristic we can discover the bug in the motivating example within the first ten iterations.

5. IMPLEMENTATION

We have implemented the test_reduce algorithm of Section 4 as a wrapper around our Dsc dynamic symbolic execution framework [6]. As Dsc analyzes Java programs, our implementation is currently targeted at programs written for the popular Java open source MapReduce system Hadoop.

The MapReduce programmer invokes test_reduce, maybe as part of a continuous automated testing process. Our implementation works by instrumenting the original user MapReduce program at the bytecode level at load-time [2]. This allows our implementation to run on any standard Java virtual machine.

6. RELATED WORK

In [7], an algorithm is proposed to generate small and yet real example data for examining the behavior of dataflow programs. Their technique is based on nonuniform data sampling combined with data synthesis. Their goal is not to directly capture program bugs but to use small example data to make programs exhibit behavior similar to those exposed by the real data. Specifically, the programmer must define the semantics of query operators that are to be comprehensively covered by the small example data. Mochi [12] visualizes execution logs to debug Hadoop programs. The focus of this tool is on execution performance anomalies instead of correctness of programs. Sen and Agha use dynamic symbolic execution for testing of distributed programs [10].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1017305.

7. REFERENCES

[1] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th International SPIN Workshop on Model Checking of Software*, pages 2–23. Springer, Aug. 2005.

[2] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with Joie. In *Proc. USENIX Annual Technical Symposium*, pages 167–178. USENIX, June 1998.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150. USENIX, Dec. 2004.

[4] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 59–72. ACM, Mar. 2007.

[6] M. Islam and C. Csallner. Dsc+Mock: A test case + mock class generator in support of coding against interfaces. In *Proc. 8th International Workshop on Dynamic Analysis (WODA)*, pages 26–31. ACM, July 2010.

[7] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *Proc. 35th ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 245–256. ACM, June 2009.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proc. 34th ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110. ACM, June 2008.

[9] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, Oct. 2005.

[10] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Proc. 9th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 339–356. Springer, Mar. 2006.

[11] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: Friends or foes? *Communications of the ACM*, 53(1):64–71, Jan. 2010.

[12] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Mochi: Visual log-analysis based tools for debugging Hadoop. In *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, June 2009.

[13] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endowment*, 2(2):1626–1629, Aug. 2009.

[14] T. White. *Hadoop: The definitive guide*. O'Reilly, May 2009.