OptDebug: Fault-Inducing Operation Isolation for Dataflow Applications

Muhammad Ali Gulzar
Virginia Tech
gulzar@cs.vt.edu

Miryung Kim
University of California , Los Angeles
miryung@cs.ucla.edu

ABSTRACT

Fault-isolation is extremely challenging in large scale data processing in cloud environments. Data provenance is a dominant existing approach to isolate data records responsible for a given output. However, data provenance concerns fault isolation only in the data-space, as opposed to fault isolation in the code-space—*how can we precisely localize operations or APIs responsible for a given suspicious or incorrect result?*

We present OptDebug that identifies fault-inducing operations in a dataflow application using three insights. First, debugging is easier with a small-scale input than a large-scale input. So it uses data provenance to simplify the original input records to a smaller set leading to test failures and test successes. Second, keeping track of *operation provenance* is crucial for debugging. Thus, it leverages automated taint analysis to propagate the lineage of operations downstream with individual records. Lastly, each operation may contribute to test failures to a different degree. Thus OptDebug ranks each operation's *spectra*—the relative participation frequency in failing vs. passing tests. In our experiments, OptDebug achieves 100% recall and 86% precision in terms of detecting faulty operations and reduces the debugging time by 17× compared to a naïve approach. Overall, OptDebug shows great promise in improving developer productivity in today's complex data processing pipelines by obviating the need to re-execute the program repetitively with different inputs and manually examine program traces to isolate buggy code.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Information systems** → *MapReduce-based systems.*

SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3487016

KEYWORDS

data intensive scalable computing, debugging, bug isolation, taint analysis

ACM Reference Format:
Muhammad Ali Gulzar and Miryung Kim. 2021. OptDebug: Fault-Inducing Operation Isolation for Dataflow Applications. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3487016

1 INTRODUCTION

Debugging is an inevitable part of software development. Developers typically repeat program comprehension and re-execution with different inputs numerous times over and over as they develop and maintain large software systems. The debugging process is incredibly challenging in the domain of Data-Intensive Scalable Computing (DISC) systems, such as Google's MapReduce [19], Hadoop [2], and Apache Spark [1], due to their distributed nature and complexity. DISC systems are used in almost every production software, as the data processing needs often exceed petabyte-scale now. When writing dataflow applications, developers first express custom logic by declaring user-defined functions and by stitching them with dataflow APIs, such as `map` and `reduce`, in mainstream languages such as Scala or Python. Debugging the root cause of a wrong output, a crash, or slow performance is difficult due to the lack of transparency and visibility into computation in DISC systems.

Prior work on Data Provenance (DP) can help with debugging faulty data in the dataflow application [28, 32]. DP keeps track of the lineage between an individual input record, its intermediate record, and eventually the final output record. Thus, DP can be used for explaining how a dataflow application produces a given output by returning a minimal subset of input records needed to generate that output. Despite low overhead and fast provenance query speed [27], DP techniques are limited in identifying culprit records in the data-space only, as opposed to identifying faulty logic or operations in the code space. Therefore, when the input data is considered clean and when a user must identify faulty code or operations leading to an undesirable outcome, a developer must spend significant time re-executing code with different inputs and analyzing their execution paths.

In the software engineering community, there exists a broader category of code debugging techniques called, spectra-based fault localization (SBFL) [29]. When a set of passing tests and failing tests are available, SBFL collects program traces and contrasts the differences in the coverage profile (*i.e.,* exercised statements or paths), and assigns a *suspicious score* to each exercised statement based on its degree of contribution to passing vs. failing tests.

However, it is not straightforward to apply SBFL to DISC applications for two reasons. First, a DISC application by nature takes extremely large data as input (*i.e.,billions of records*). Without simplifying the original data to a smaller scale, storing a program trace for each input record during distributed, parallel execution would increase both the memory footprint and runtime overhead. Second, to capture statement coverage, SBFL must deploy an instrumented JVM (or a code coverage tool) in all remote nodes.

To address this challenge of *automated code debugging* of DISC applications, OptDebug employs a novel operation-level taint-analysis to determine the root cause code fault precisely without high-overhead, intrusive instrumentation. OptDebug takes as inputs—a DISC application, a test predicate, and an input dataset—and returns a ranked list of operations (*i.e.,* a line number in code with a primitive operation). For each attributed line, it also reports a *suspicious score* to quantify how likely that code line contributes to test failures.

The key insights behind OptDebug is two folds. First, it simplifies the data debugging problem through data provenance. It turns a massive data set into a smaller, manageable data set that is capable of reproducing test failures or successes. Second, it solves the code debugging problem with this reduced data set by redesigning taint analysis to work at an operation level, so that it tracks *the history of applied operations* as opposed to the history of affected data. Currently, OptDebug supports Apache Spark applications with operation tainting at two granularities: line of code and APIs for primitive types such as String's `split`. The underlying methodology can be adapted to different DISC platforms.

We evaluate OptDebug on three fronts: (1) debugging time speed up, (2) accuracy of fault localization in terms of both precision and recall, and (3) runtime overhead. We design experiments on a diverse set of DISC application benchmarks adopted from prior work [22, 23, 40], running on a 104 cores 13-node cluster. Across six subject applications, OptDebug takes only 47% of the original job time to complete the automated debugging process. This is a significant 17× speed up over the baseline debugging via taint analysis, showing the impact of input reduction by data provenance on OptDebug's debugging process. OptDebug is capable of identifying fault-inducing code operations with 86% precision and 100% recall. The primary reason for such high accuracy and the significant speedup is that operation-level taint analysis is enabled for a much smaller set of curated inputs obtained through data provenance. Compared to the traditional SBFL-based approach, OptDebug takes 27% less time on simplified input from data provenance. We further assess the effectiveness of different suspicious scores: Ochiai [9], Tarantula [29], Barniel [8], and OP2 [35]. Tarantula score shows the most promising fault localization accuracy with 86% precision, while Ochiai, OPT2, and Barniel report 44%, 25%, and 80% precision respectively.

We summarize the contributions of this paper below:

- To our knowledge, OptDebug is the first fully automated *code debugging* tool for DISC applications. It goes beyond prior work that enabled *data debugging* only.
- OptDebug proposes a novel operation-level taint analysis to track the history of executed code lines and APIs. This taint tracking analysis does not require modifications to runtimes and thus is easy to deploy.
- OptDebug is the first to embody spectra-based fault localization for DISC application debugging and to evaluate various *suspicious scoring methods* systematically.
- OptDebug identifies faulty operations very quickly. By paying an upfront overhead of data provenance ranging from 1.1× to 5×, a user can precisely identify fault-inducing operations in under 47% of the job execution time on vanilla Spark, leading to significant time savings.

By realizing OptDebug in a real-world tool, we show that input simplification of data provenance can be combined with fine-grained taint analysis to perform code debugging. OptDebug is open source and publicly available at `https://github.com/maligulzar/OptDebug`. We organize the rest of the paper as follows. Section 2 presents a motivating example that highlights debugging challenges. Section 3 introduces OptDebug and describes the key insights and its approach. Section 4 presents the results from our extensive set of experiments on OptDebug. Section 5 sheds light on related work in this area and how OptDebug compares to it. We conclude this paper in Section 6.

2 MOTIVATING EXAMPLE

This section presents a running example to emphasize the motivation behind OptDebug and highlight its benefits. We take inspiration from a benchmark in prior work on data provenance [23]. OptDebug is built on top of Apache Spark for Scala programming language but its approach is applicable to other data processing frameworks or programming languages with data provenance support.

```
1  val log = "s3://IATA-data/logs-2020/transit.log"
2  val input = new SparkContext(sc).textFile(log)
3  input.map { s =>
4    val tokens = s.split(",")
5    val dept_hr = tokens(2).split(":")(0)
6    val diff = getDiff(tokens(4), tokens(2))
7    (dept_hr, diff)
8   }
9   .filter(v => v._2 < 4)
10  .reduceByKey(_+_)
11
12 // Calculates the difference between time
13 def getDiff(arr: String, dep: String): Float = {
14  val arr_hr =  arr.split(":")(0).toFloat  + arr.
      split(":")(1).toFloat/60
15  val dep_hr =   dep.split(":")(0).toFloat + dep.
      split(":")(1).toFloat/60
16  if( arr_hr - dep_hr < 0){ // across midnight
17    return arr_hr - dep_hr - 24
18  }
19  return  arr_hr - dep_hr
20 }
```

Figure 1: A dataflow application in Apache Spark that takes an airline transit dataset as input and calculates the total flying hours for less-than-four hour flights grouped by each departure hour.

Suppose Alice is a data scientist who is given the task of measuring the total flying hours for all flights with less than 4 hours grouped by their departure hour. The entire dataset is several gigabytes in size and contains the telemetry data from millions of flights flown by airlines over the past 20 years. Every row in the dataset follows a CSV (comma-separated value) format. The first column represents the 9-letter alpha-numeric Flight ID. The second column represents the departure airport code. The third column represents the departure time in UTC 24-hour format, and the fourth column represents the arrival airport code. The last value represents the arrival time in UTC 24-hour format. For instance, the following record shows the flight departed from Charlotte, NC (CLT) at 23:15 UTC and arrived in Atlanta, GA at 0:15 UTC.

```
AAB141715 , CLT , 23:15 , ATL , 0:15
```

Alice writes a dataflow application in Apache spark shown in Figure 1. Line 2 reads the dataset stored in Amazon S3 storage using the `textfile` API and applies a `map` operator to extract the departure hour as key (line 5). Line 6 computes the difference between the departure and arrival time. Line 9 filters out the flights with a flying time greater than 4 hours. Line 11 aggregates the total flight time per key group. Alice submits this application on a public cloud platform provisioned with 32 instances to run her analysis on the entire dataset. After several minutes, the application returns the following results.

```
(11 , 175080)
(20 , 173460)
(23 ,-222780)
```

Categorizing Suspicious Outputs. By skimming over a few output records, Alice notices a negative flight duration, a suspicious or incorrect output. For example, the total duration for the flights departed at 23:00 is -222780. Such incorrect individual output can be seen as a test failure. In fact, Alice can write the following test predicate (a boolean function) to check whether each output record is correct:

```
def test(output: (String, Float)) : Boolean =
 output._2 > 0
```

To determine the root cause of such bug, Alice may attempt two different debugging directions: **(1) data debugging:** *which subset of input records contributes to the incorrect output records?* and **(2) code debugging:** *which subset of operations (i.e., code line or API) in her application is responsible for the incorrect output records?* Alice may use data provenance tools for data debugging and use spectra-based fault localization tools for code debugging, each of which has limitations in the DISC application domain.

Limitations of Data Provenance. To find a subset of input records leading to suspicious or incorrect outcomes, Alice may use data provenance tools such as Titian [27] or Smoke [38]. She enables Titian on her application and invokes a backward tracing query on each negative flight duration output. Titian returns a subset of 2 million records (approximately 0.2% of the original input data), corresponding to the key 23.

```
XAY993311 , CLT , 23:15 , ORD , 1:15
EWS121311 , LAX , 23:45 , ORD , 5:00
AAQ591783 , SJC , 23:33 , MNN , 4:20
```

The fault-inducing input records seemingly look clean, valid, and error-free. At this point, Alice suspects that faults are caused by *code* not *corrupted data.* If she re-executes the program for almost two million input records and manually inspects generated program traces, it would be a painfully time-consuming process to detect code faults.

Limitations of Spectra-based Fault Localization. One dominant approach for code debugging is to use spectra-based fault localization techniques such as Tarantula [29] and Ochiai [9]. They assume a high-quality test suite with inputs leading to both test successes and failures. For instance, Tarantula measures a suspicious score based on the execution count of each code line in failing vs. passing tests. Such a technique is not readily applicable to DISC applications. Existing coverage tools perform virtual machine (*e.g.,* JVM) level native instrumentation, which must be deployed at each

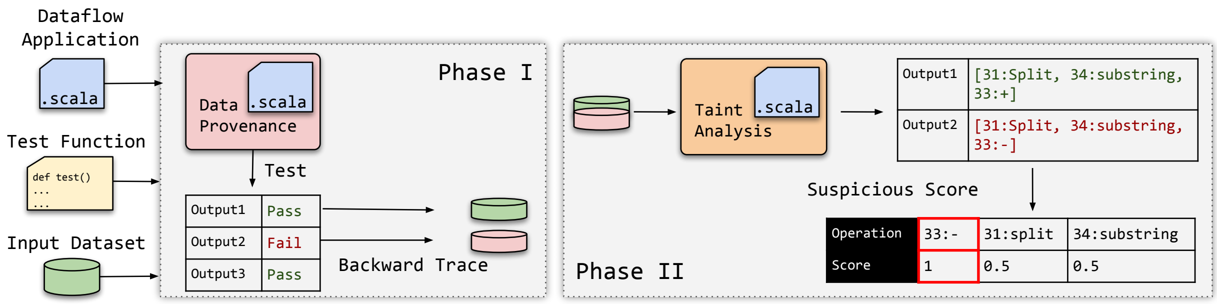

Figure 2: Phase I takes as inputs—a dataflow application, a test predicate, and an original input data. It then applies data provenance to simplify inputs, leading to passing vs. failing outcomes w.r.t the test predicate. Phase II enables taint analysis for the simplified input and ranks the contributing operations in terms of suspicious scores.

```
1   val log = "s3://IATA-data/logs-2020/transit.log"
2 - val input = new SparkContext(sc).textFile(log)
3 + val input = new OptDebug(sc).textFile(log)
4   input.map {..}
5      .reduce{..}
6 -    .collect()
7 +    .runWithOptDebug[(String, Float)](
            test = r => r._2 >0)
```

Figure 3: A user can adapt their application code with a two line modification (lines 2 and 6) to enable OptDebug.

worker node in the cloud environment. Such an approach poses an excessive overhead of executing instrumented code and storing program traces from distributed, remote nodes. Furthermore, to distinguish the code coverage of each test, SBFL would need to execute the application for each input in isolation repetitively.

In summary, even if a code coverage collection tool can be enabled for distributed, parallel execution, it would pose significant scalability and performance challenges due to execution overhead, repetitive runs, and coverage collection overhead from remote nodes. Therefore, SBFL is currently not feasible for DISC applications.

OptDebug for Automated Code Debugging. To use OptDebug, Alice needs to make a small edit to her code by adding `new OptDebug()`, as shown in Figure 3. OptDebug then automatically turns on data provenance for the first run and executes the application with the entire input data. It then uses the user-defined test predicate to classify which output records are failing vs. passing and invokes a backward tracing query for each failing output. With this returned input subset, OptDebug turns on operation-level taint analysis and re-runs the application for the second run. OptDebug collects the history of applied operations and covered code lines and uses spectra-based fault localization techniques to rank faulty code.

As a result, OptDebug reports culprit buggy code lines and operations, arranged in the descending order of suspicious scores. OptDebug takes 8 seconds for tainting-enabled execution for the second run on the substantially reduced data set from the first run:

```
(Line: Operation, Score)
(17: Int.new(), 1.0)
(17: Int.minus, 1.0)
(15: String.split, 0.47)
(14: Int.times, 0.47)
```

OptDebug reports a suspicious score of `1.0` to line 17's `new Int` and `minus` operations. With this information, Alice immediately realizes that, at line 17, the value of 24 hours should be added instead of subtracted, which gives the very hint needed for fixing her program.

3 APPROACH

To determine the root cause of coding errors, OptDebug takes the user application, a test predicate, and the original input data as inputs and reports a ranked list of operations with a corresponding suspicious score. OptDebug employs a two-phase approach, shown in Figure 2.

Phase I enables data provenance (provided by Titian [27]), executes the program on the entire input dataset to capture data lineages, classifies incorrect output records using the test predicate, performs backward tracing on a select subset of failing vs. passing outputs, and discards the rest. The returned subset of input records is often substantially smaller than the original input data, yet they can trigger both failing and passing behavior.

Phase II re-executes the job with this reduced data. This time, the application is automatically refactored to perform operation-level taint analysis to track the history of applied operations to each input record. OptDebug applies SBFL models to calculate a suspicious score for each profiled operation. Operations with the highest suspicious score are returned to the user as the root causes.

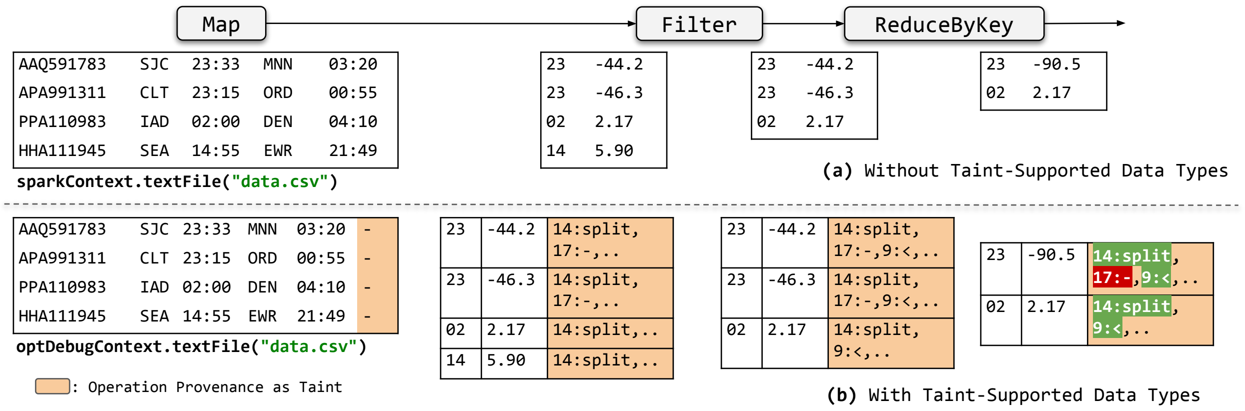
Map
Filter
ReduceByKey
AAQ591783 SJC 23:33 MNN 03:20
APA991311 CLT 23:15 ORD 00:55
PPA110983 IAD 02:00 DEN 04:10
HHA111945 SEA 14:55 EWR 21:49
sparkContext.textFile("data.csv")
(a) Without Taint-Supported Data Types
optDebugContext.textFile("data.csv")
14:split, 17:-,..
14:split, 17:-,9:<,..
14:split, 17:-, 9:<,..
: Operation Provenance as Taint
(b) With Taint-Supported Data Types

Figure 4: Phase II illustration with the example from §2. (a) execution without taint analysis. (b) with taint analysis. Orange boxes represent operations tagged onto individual records through taint analysis. As opposed to traditional taint analysis that propagates the origin data, OptDebug propagates code lines and operations onto each record. At the end of the pipeline, red and green mark success-inducing vs. failure-inducing operations with a corresponding line number respectively.

3.1 Phase I: Simplifying Input via Data Provenance

The goal of Phase I is first to solve the *data debugging* problem: simplifying the original input data to a substantially reduced input data. This reduced data is related to a small number of success and failure outcomes, defined via the given test predicate.

Categorizing Output Records with User-Defined Test Predicate. OptDebug uses the user-provided test predicate to classify individual output records. OptDebug then *randomly* selects the p and f number of passing and failing outputs respectively. Default p and f are set to 2. The implication of random selection and how many p passing and f failing outputs to retain is discussed in Section 4.4.

Input Reduction via Data Provenance. OptDebug uses data provenance by Titian [27], an interactive data provenance tool built for Apache Spark. It enables backward and forward data tracing query support with reportedly 30% runtime overhead to capture the relationship between inputs, intermediate outputs in each stage, and final outputs. Titian creates such lineage tables at each shuffle boundary. Each intermediate input (/output) is tagged with a provenance ID, and the input and output mappings are stored in the in-memory storage for fast, recursive, distributed join queries required for tracing. Details on Titian are described else where [27, 28].

OptDebug uses Titian's backwards tracing query, `lineage.filter(P).goBackAll` on the selected p+f number of passing and failing outputs. Behind the scenes, Titian recursively joins the lineage tables one by one, starting from each output and working its way towards the input. At the end of tracing, Titian returns a significantly smaller set of inputs (on average four orders of magnitude, 10^{-4}, smaller than the original entire data set). By removing a majority of data records from the input subset that do not add additional value to the SBFL process, this reduction directly impacts Phase II's performance, as evaluated in Section 4.

3.2 Phase II: Spectra Debugging via Operation-Level Taint Analysis

Phase II extends taint analysis to capture the provenance of operations for each output data record using the simplified reduced input. It then measures a suspicious score for each operation using spectra-based fault localization by contrasting its likely contribution to passing vs. failed outcomes.

Traditional SBFL-based techniques require code coverage tools to monitor statements executed by each test case. Each test must be executed in isolation to create an independent coverage profile. To achieve this in DISC systems, we must deploy a cluster-wide JVM instrumentation and run each input record in isolation. Even if this is feasible with the simplified input from Phase I, collecting coverage from repeated runs and across multiple remote nodes would be prohibitively expensive. Additionally, SBFL-based techniques collect coverage to isolate all code lines, and thus cannot differentiate Apache Spark implementation code from user-level application code.

For the ease of adoption and lightweight coverage collection, OptDebug extends taint analysis to capture *the history of applied code operations*, as opposed to *the origin and trace of affected data*. Compared to traditional SBFL-based techniques, OptDebug stays in application layer and requires no complex system-level modification. It eliminates the need

```
new SparkContext(sc)
 .textFile(log) :RDD[String]
 .map()         :RDD[String,Float]
 .filter()      :RDD[String,Float]
 .reduce()      :RDD[String,Float]
```

(a) RDD data types with `SparkContext.textFile()`

```
new OptDebug(sc)
 .textFile(log):OptRDD[TaintString]
 .map()        :OptRDD[TaintString, TaintFloat]
 .filter()     :OptRDD[TaintString, TaintFloat]
 .reduce()     :OptRDD[TaintString, TaintFloat]
```

(b) RDD data types with `OptDebug.textFile()`

Figure 5: OptDebug takes advantage of Scala's static type inference to incorporate tainted data types with a simple, single line code modification.

to make drastic framework or JVM changes across all nodes and does not require a separate application to consolidate coverage logs from all nodes. By keeping track of this operation provenance through application-level taint analysis, OptDebug is runtime-system agnostic and easily portable. Taking inspiration from traditional taint analysis [36] and flow tracking [40], OptDebug automatically re-writes a user application by inserting a simple API invocation that wraps `SparkContext` API with 2 lines code insertion, as shown in Figure 3.

Operation-Level Taint Analysis. OptDebug's operation taint analysis takes advantage of static type inference and operator overloading to automatically replace the used data types with the corresponding taint-supported data types. A taint-supported data type contains a reference to the original value and a set of provenance tags at the *code or operation* level. The provenance tags are then stored as a `BitSet` of line numbers in code or a `Set` of API operations.

The taint-supported data types such as `TypeInt` have the same exact signature, methods, and operations as their original counterpart such as `Int`. For example, in Figure 1, lines 3-8 represent a UDF that takes a `String` and outputs a tuple of `String` and `Float`. While the UDF is meant to take `String` as input, OptDebug can reuse the same UDF at compile time, and assume the input type as `TaintString` which mirrors all `String` operations. Therefore, the UDF type-checks with `TaintString` as input and `(TaintString, TaintFloat)` as output. Similarly, OptDebug can change the entire dataflow application's data type to a taint-supported data type by simply making a single line code change in the beginning of the application: re-writing the original input loading procedure, `rdd.textFile()` to return `OptRDD` instead of `RDD`—Resilient Distributed Datasets (RDD) is a fundamental data structure in Spark, an immutable distributed

```
1  case class TaintInt(value: Int){
2
3   // Callee line number and operation name
4   setProvenance(getCallSite())
5
6   def +(x: Int): TaintInt =
7    TaintInt(value + x, newProvenance(getCallSite())
     )
8
9   def +(x: TaintInt): TaintInt =
10   TaintInt(value + x.value,
11     mergeProvenance(this,x.provenance,
12       getCallSite()))
13 ...
```

Figure 6: Each operator (or API) is instrumented to support taint analysis in OptDebug. In Line 9, `TaintInt` **overloads the binary operator + with** `TaintInt` **for the right and left parameters.** `getCallSite()` **returns a provenance object containing the callee's line number and the current operation name.**

collection of objects. This type change has the effect of using an `OptRDD` of type `TaintString` instead of an `RDD` of type `String`, which initiates taint tracking. Figure 5 shows how `RDD` types change when `OptDebug` is used instead of `SparkContext` as an input loading procedure.

Figure 4 illustrates the operation-level taint analysis on the example from Section 2. Each operation in a taint-supported data type internally adds the name of the operation and the callee's line number to the existing provenance set. This step of retaining both the operation and the corresponding line number is critical for fine-grained debugging, because the same operation may be called from multiple lines, and a single code line may involve multiple operations. In Figure 4 (b), the use of a binary comparison operator '<' at line 9 `v => v._2 <4` in Figure 4 would invoke a corresponding taint-supported operation '<' in `TaintInt`, in turn adding the provenance tag of `9:<` (*i.e.,* line 9 uses a binary operator '<') after the `Filter` transformation. Similar to the typical taint analysis, the interaction between two provenance tags leads to the union of both provenances. For example, at line 19 in Figure 1, `arr_hr - dep_hr` involves two `TaintFloat` variables hence the minus operator merges the provenance tags of `dep_hr` with the provenance of `arr_hr`.

Currently, OptDebug supports operation-tainting for all primitive types such `String`, `Float`, `Int`, `Array`, and `Int` and frequently used Java libraries such `Math`. Figure 6 shows the implementation of `TaintInt`. Similar to typical taint analysis, developers do not need to define taint-supported types themselves, as they can be defined once for all commonly used types.

Spectra-based Code Debugging. Spectrum-based fault localization (SBFL) records the execution coverage profile, most commonly at the line level, of a program for each test input. It then contrasts each line's contribution to multiple failing vs. passing tests to determine and rank suspicious code regions, more likely to be associated with failing tests but not passing tests. The key assumption is that a code line is more suspicious if it participates in failing traces but not in any passing traces. Suppose we have a test suite that consists of three test cases: $[t1_{pass}, t2_{fail}, t3_{pass}]$, where *pass* and $fail$ represents the test outcome and that the corresponding coverage profile is $[(2, 4, 5), (2, 3, 4), (2, 4, 5)]$ respectively, where $(2, 4, 5)$ means $t1$ exercises lines 2, 4, and 5. Line 3 is invoked by a failing test and not any of the passing tests, indicating a high likelihood of being the root cause. Recent work on SBFL proposes various methods to calculate the suspicious score of each code line [37].

After the second run with taint analysis, OptDebug collects output records, each carrying their respective provenance tags. It re-applies the user-provided test predicate to distinguish which provenance tags contribute to passing vs. failing outcomes. To calculate a suspicious score, OptDebug measures the following metrics:

- *totalfail*: the total number of failing outputs
- *totalpass*: the total number of passing outputs
- *fail(op)*: the number of failing outputs whose provenance tag contains the given operation, *op*.
- *pass(op)*: the number of passing outputs whose provenance tag contains the given operation, *op*.

Ranking based on Suspicious Score. OptDebug adopts various suspicious score computation methods from the SBFL literature, including Tarantula [29], Ochiai [9], Barniel [8], and OP2 [35]. By default, OptDebug uses Tarantula to rank operations, defined as follows:

$$Tarantula(op) = \frac{\frac{fail(op)}{totalfail}}{\frac{fail(op)}{totalfail} + \frac{pass(op)}{totalpass}} \tag{1}$$

Other suspicious score computation methods are defined as:

$$Ochiai(op) = \frac{fail(op)}{\sqrt{totalfail \cdot (fail(op) + pass(op))}} \tag{2}$$

$$OP2(op) = fail(op) - \frac{pass(op)}{totalpass + 1} \tag{3}$$

$$Barniel(op) = 1 - \frac{pass(op)}{pass(op) + fail(op)} \tag{4}$$

Phase II returns a ranked list of operations in the descending order of suspicious score. The code line or APIs with the highest suspicious score is the most likely to be the root cause, in other words, a fault-inducing operation.

3.3 OptDebug's API

OptDebug is a Scala library (jar file) that can easily be imported into any Apache Spark application. It wraps the entry point of Apache Spark *i.e.,* `SparkContext` with `OptDebug`.

```
class OptDebug(sc:SparkConf) {
 def textFile(String:log) : OptRDD[TaintString]
...
class OptRDD[U] {
 def runWithOptDebug[U](test: U => Boolean)
...
```

A user can initiate taint-tracking with simple two line modifications. While loading an input to create an RDD, she can invoke the `OptDebug.textFile()` API instead, which returns an `OptRDD` of type `TaintString` instead of an `RDD` of type `String`. A user can then invoke `runWithOptDebug` with a test, which triggers the debugging process, if the user-provided test predicate fails on any output.

3.4 Implementation

OptDebug is written in Scala programming language and packaged into a jar file. It relies on two key language features: static type inference and operator overloading, both supported in Scala and Python, the two widely used languages in data analytics. Behind the scenes, OptDebug requires a data provenance tool in Phase I. Its current implementation uses Titian [27], but it can be easily ported to use a more recent data provenance tool. OptDebug's taint analysis is completely runtime-agnostic, as it does not require modifications to a DISC system runtime and is implemented as an application-level analysis (*i.e.,* Python or Scala frontend).

4 EVALUATION

We evaluate OptDebug on three key criteria *i.e.,* debugging time, fault localization accuracy, and runtime overhead. Specifically, we aim to answer the following research questions:

- **Fault Localizability:** What are OptDebug's precision and recall in localizing fault-inducing code or operation?
- **Debugging Time**: How long does OptDebug take to find fault-inducing code in a user application?
- **Runtime Overhead**: What is the runtime overhead of OptDebug?

We perform a series of systematic experiments on a diverse range of subject program benchmarks running on a large-scale cluster environment.

ID	Program	Description	LOC	Dataset Size	Program Faults	Source
P1	Flying Hours	Find the total flying hours per departure hour	25	25GB	Error in time calculations	Gulzar et al. [23]
P2	Commute Mode Trips	Estimate the total trips per mode of transport	50	2.4GB	Wrong threshold for "on foot" trips	Gulzar et al. [22]
P3	Movie Age on Netflix	Find the count of movies per movie age on Netflix	40	28GB	Age calculation fault for movies before 2000	Kaggle Netflix Dataset [5]
P4	Student GPA	Find each student's GPA from course scores	45	93GB	Typo in percentage to GPA conversion	Teoh et al. [40]
P5	US Car Accidents	Find the average visibility level per accident severity type	35	22GB	NaN visibility measurement mapped to zero	Kaggle US Accident Dataset [5]
P6	Weather Analysis	Find the delta between max and min snowfall reading per state per month	65	37GB	Measurements in "inches" treated as "mm"	Gulzar et al. [20]

Table 1: Subject Programs, Data Sets, and Program Faults

Subject Programs. Real-world applications for DISC systems are mostly closed-source with closed data and they do not come with known code faults. Therefore, there is a lack of applications on which OptDebug can be evaluated. Existing DISC benchmarks, such as TPC [7], HiBench [3], and PigMix [6], are performance benchmarks with no real bugs and with minimal programming logic diversity.

To evaluate OptDebug, we select six subject dataflow applications with known faults. These programs ingest real-world data as input and comprise a diverse set of dataflow operators with minimal logical overlap between individual applications. Specifically, the programs leverage frequently used operators such as `map`, `flatmap`, `averageByKey`, `reduceByKey`, `groupByKey`, and `filter`. The faults in these program are inspired by bugs reported on Stackoverflow/-mailing lists [22, 44]. The benchmark programs are publicly available at `https://github.com/maligulzar/OptDebug`.

Table 1 lists the six subject programs and their sources. Program P1, P2, and P6 are derived from related work on DISC application debugging and test input generation [20, 22, 23]. These three programs come with pre-injected code faults in them. Program P4 is a slightly modified version of a benchmark presented elsewhere [40]. Programs P3 and P5 represent data analytics tasks performed on Kaggle's open Datasets [5]. For those programs without known code faults (P3, P4, and P5), we systematically inject faults based on Zhang et al.'s survey of commonly found errors in big data applications [44]. All subject programs come with large-scale datasets ranging from 2.4 GB to 93GB. For Kaggle datasets, we scale up the dataset size from a few GBs to tens of GBs via duplication to evaluate OptDebug's scalability.

Experiment Environment. We perform our experiments on a dedicated large-scale cluster computing environment that spans 13-nodes where each node has 8 cores, 4TB storage, 3.10GHz CPU, and 48GB memory. Overall, the cluster offers approximately 104 cores, 53TB disk storage, and 576GB memory. We deploy Apache Spark 3.0.1 along with Apache Hadoop HDFS 2.7. The replication factor for HDFS is set to 3. During our experiments, we dedicate the entire cluster to HDFS and shut down other services to minimize the performance impact of other applications on the evaluation results. We restart `SparkContext` for each experiment instance to avoid any implicit caching. Every experiment is repeated four times, and the results are averaged to balance out any outlier.

Baseline. We compare OptDebug's debugging accuracy using standard metrics of precision (*how many of* OptDebug*'s results are true bugs?*) and recall (*how many of known fault-inducing operations are found by* OptDebug*?*). For debugging time, we use two baselines. First, we compare OptDebug with an operation-level taint analysis technique without data provenance. This baseline will enable operation-tainting for the entire input data, as opposed to simplified input data. Second, we conservatively construct a baseline where OptDebug's taint analysis is replaced with standard line coverage collection using the IntelliJ Idea coverage plugin. The reason behind having two baselines is to assess the benefits of each insight *i.e.,* (1) using data provenance to simplify an input data set, and (2) using taint analysis to keep track of operation instead of coverage profiling per test. For overhead evaluation, we compare the execution time of the data provenance job using Titian and the original job using vanilla Spark.

4.1 Fault Localizability

To evaluate OptDebug's capability to detect code faults, we measure how precisely and accurately OptDebug finds faulty code lines (/operations) in the subject programs. While OptDebug reports potential faults in a ranked list, in our evaluation, we conservatively assume that a developer may consider only the code faults with the highest suspicious score. The number of code faults with the highest suspicious score can be more than one when there is a tie.

Program	Input Count	Simplified Input via Titian	Known Faults	Detected Faults
P1	8.8 x 10^8	1,730,183	2	2
P2	9.6 x 10^7	45,515	1	2
P3	7.8 x 10^7	220,000	2	3
P4	5.0 x 10^9	198,000	1	1
P5	5.7 x 10^7	210	1	1
P6	1.6 x 10^9	709,200	2	2

Table 2: Input Reduction by Data Provenance. Known Faults vs. Detected Faults

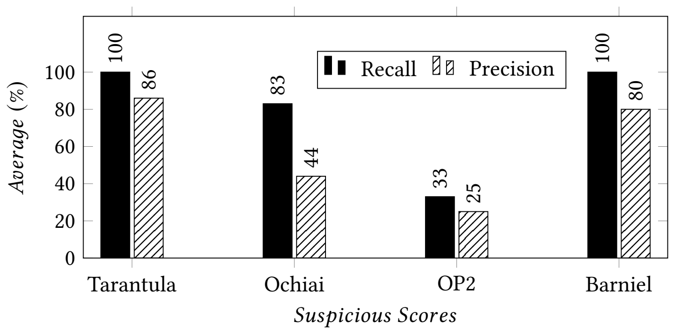

Figure 7: OptDebug**'s average precision and recall with various suspicious score calculation methods.**

For each fault, we use the operation name and the corresponding line number as a ground truth. In terms of suspicious score, we use Tarantula as a default method described in Equation 1. In Table 2, column **Known Fault** shows the number of known faults present in each program, and column **Detected Faults** shows the number of faults that OptDebug reported with the highest suspicious score. For instance, in program P2, an incorrect speed threshold, (10, 15) instead of (0, 15), is used to classify trips covered on foot.

```
if (10<speed<15) //  Correct: 0<speed<15
    ("Foot", speed)
else
    ("Invalid", speed)
```

OptDebug returns the ranked list of operation with their suspicious scores: `[(77 : >, 1.0), (80 : Int.new(), 1.0), (67 : >, 0.15), ...]`. OptDebug identifies the two locations (lines 77 and 88) with the highest, equal suspicious score. `(77: >)` is the location of the actual, injected fault in `if` condition. `(80 : Int.new())` is a false positive, as it corresponds to the `else` block. On P2, OptDebug shows the recall and precision of 100% and 50% respectively. In contract, on program P1, OptDebug isolates the single most precise operation responsible for a test failure, reaching 100% precision and recall.

To understand the efficacy of various suspicious score calculation methods, we compare the precision and recall of OptDebug with different SBFL methods *i.e.,* Tarantula [29], Ochiai [9], OP2 [35], and Barniel [8]. Figure 7 presents the average precision and recall for each method across six programs. Y-axis represents recall and precision, and X-axis represents a different method. When using Tarantula, OptDebug achieves 86% precision and 100% recall on average, performing better than the other three. In contrast, OP2 performs the worst with only 33% recall and 25% precision. Overall, Tarantula seems to be the best choice for OptDebug hence set as a default method.

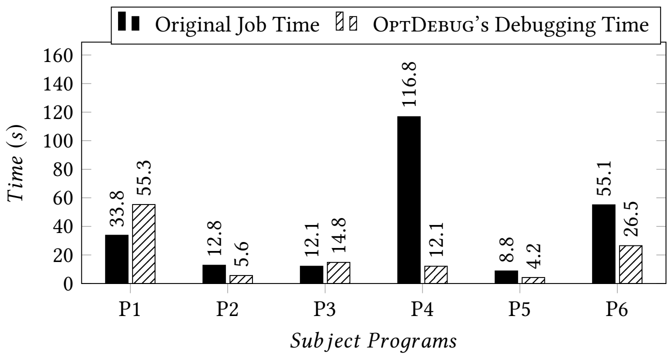

Figure 8: OptDebug **running time is on average 47% faster than the time to execute the target program with the original, entire input, showing significant speed up over trial and error debugging.**

4.2 Debugging Time

We evaluate the time OptDebug takes to find the fault-inducing operation after a given application produces a failing outcome defined via a test predicate. We perform three different kinds of comparison in order to shed light on where and how OptDebug can save debugging time:

- OptDebug's total debugging time vs. the original job execution time: A developer tends to re-execute a program over and over again during trial and error debugging. Therefore, OptDebug's total debugging time should be much faster than the original job time.
- OptDebug's debugging time with and without taint analysis: This experiment is to assess the benefit of operation-tainting, as opposed to leveraging a code coverage collection tool to gather the coverage profile of each input in isolation and to apply SBFL.
- OptDebug's debugging time in Phase II with the simplified input vs. the entire input (*i.e.,* with and without data provenance): This experiment is to contrast the role of solving *data debugging* prior to solving *code debugging* through input simplification.

Debugging time vs. the original job time. We compare the total debugging time taken by OptDebug against the original

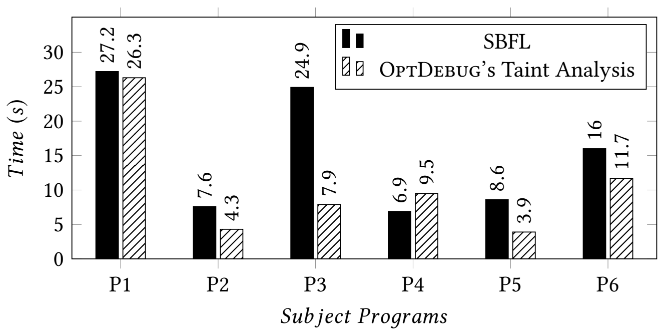

Figure 9: OptDebug's taint analysis on a simplified input is on average 27% faster than applying spectra-based fault localization on a simplified input.

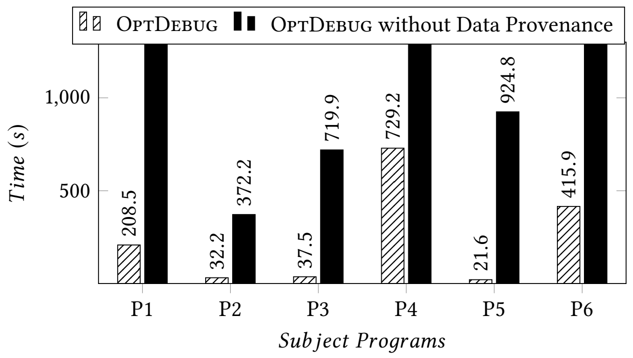

Figure 10: Simplifying the input data using data provenance can speed up debugging by 17× on average.

job execution time running on vanilla Apache Spark without any instrumentation. This is to put the total debugging time in the context of the original execution, as developers often tend to rerun the program to reproduce a failure symptom during trial and error debugging. Figure 8 shows the comparison results. Consider program P6. The original job on the entire input dataset takes 55.1 seconds on vanilla Apache Spark. OptDebug only takes 26.5 seconds (47% of the original job time) to localize code faults automatically. Program P2, P4, and P5 follow a similar trend where the total debugging time is only a fraction of the original job time. In practice, debugging a DISC application is a highly time-consuming process, with repetitive reruns of the job with the original input and different input subsets. Therefore, it is highly likely that a developer would spend 10× or 100× in debugging time, relative to the original job time. This major time saving comes from the input reduction in Phase I. By solving the data debugging problem first using data provenance, OptDebug eliminates input records that are either irrelevant or have no-additional influence on the debugging process in Phase II. Table 2 reports the reduction in the input size, which manifests into the low debugging time.

For programs P1 and P3, the debugging time is higher than the original time, 1.6× and 1.2×, respectively. For these programs, DP has a limited input simplification effect, *e.g.,* in P1 in Table 2 where the simplified input size is still in millions of records. In such cases, Phase II is slow, as taint analysis is a computation- and memory-intensive process; tainting must maintain a taint object which grows in size as the corresponding data record propagates through downstream dataflow operators. Across six subject programs, OptDebug takes only 47% of the original job time on average.

Operation-level tainting vs. coverage collection then SBFL. For Phase II, we implement operation-level taint analysis instead of the typical SBFL process that collects the coverage profile of each test case in isolation. Because the size of simplified input is still large, repetitive program execution of a single test case at a time would require significant time. Besides, coverage collection often works at the granularity of statements (or lines); thus, it cannot distinguish multiple operations on the same statement.

We compare OptDebug's operation-level taint analysis that requires only one execution with consolidated test inputs. For the baseline of coverage collection and then SBFL, we assume multiple independent test executions with coverage collection. By default, OptDebug reports code faults with the level of a line number and a corresponding operation name, which is by construction more precise. However, for a fair comparison, we downgrade OptDebug to report only the line number without the operation's name to compare OptDebug against SBFL built on a typical line-level coverage collection tool. We use IntelliJ Idea code coverage plugin to capture the line coverage in Scala.

In Figure 9, Y-axis represents the running time in seconds: OptDebug's taint analysis vs. coverage collection followed by SBFL. In both cases, we use the simplified input provided by data provenance in Phase I. Across all subject programs, OptDebug's taint analysis is consistently faster than SBFL on the simplified input, as the baseline SBFL requires multiple test executions (*i.e.,* $p = 2$ and $f = 2$, leading to 4 independent runs). In contrast, OptDebug combines four test inputs and invokes taint analysis as a single execution. Overall, OptDebug's taint analysis takes 27% less time than SBFL to report results at the same line granularity.

With and without input simplification. To quantify the impact of using data provenance for input simplification, we compare the end-to-end running time (*i.e.,* the original job time plus the total debugging time). If data provenance is not used in Phase I, Phase II must employ taint analysis on the entire input data. As shown in Figure 10, for programs P1, P4, and P6, OptDebug without data provenance takes in excess of one hour, in comparison to approximately 4, 12, and 7 minutes respectively with data provenance enabled for input

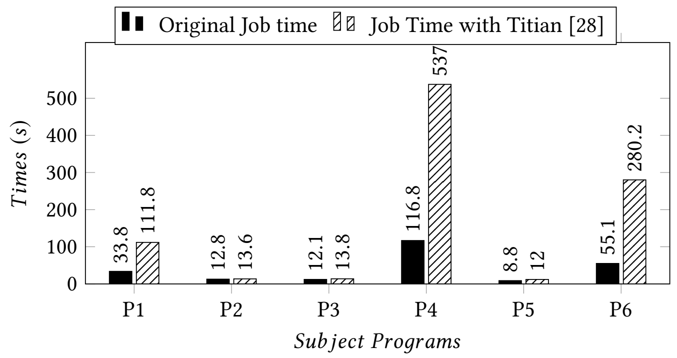

Figure 11: Runtime overhead incurred by Data Provenance, in comparison to the original job time

simplification. We set the timeout threshold on the cluster to 1 hour, after which the programs were manually terminated, as tainting takes too long. For the rest, the overhead of not using data provenance varies from 11× to 42×. The result clearly indicates the benefits of solving the *data debugging* problem first by eliminating irrelevant input records that do not contribute to the selected passing and failing tests.

4.3 Runtime Overhead

The benefit of using data provenance (DP) comes at an upfront cost of runtime overhead incurred on the original job execution. The given application must be instrumented such that the data lineage is captured at a job execution time. In the current version of OptDebug, we use Titian as the default data provenance engine (other DP tools are equally applicable). We compare the running times of a job with and without DP enabled. Figure 11 shows the time comparison between the two runs. Across six subject programs, Titian-enabled run takes from 1.1× (10% slower) to 5.1× of the original job execution. While Titian reports a 30% average overhead in their VLDB Journal publication, the dense shuffle stages (*i.e.,* the high number of unique keys with evenly distributed values) may lead to a large footprint of lineage tables and thus high overhead. Programs P2, P3, and P5 show a much reasonable overhead of up to 40%. To minimize this overhead, OptDebug may switch to leveraging more recent, advanced data tracking techniques such as Smoke [38], data influence debugging [14, 43], and input simplification [20] in addition to using Titian [27].

4.4 Discussion

Based on our evaluation, we discuss OptDebug's limitations and the future research directions to address them. We envision OptDebug as a purely debugging tool that is not intended to be enabled on production service. Based on the usage, a user may choose to enable it after observing some wrong output or keep it enabled during the program development. As mentioned before, the runtime overhead of data provenance may prohibit users from enabling OptDebug at all times. We propose using more recent low overhead data tracking tools [38] and other input simplification techniques [20]. Our decision to use Titian is solely based on its easy-to-use API and seamless integration with Apache Spark, which OptDebug currently supports. OptDebug is designed to be extensible, enabling seamless integration with any off-the-shelf data tracking tools. For instance, BigSift [20] can be used in Phase I to produce a minimal subset of records that can replicate test failures by delta debugging.

Currently, OptDebug *randomly* selects f failing outputs and p passing outputs. When using DP for input simplification, different output records may lead to different sized inputs, consequently influencing the debugging time of OptDebug. In particular, inputs with data skews are likely to result in unusually high debugging times for OptDebug, since many records with the same key may not show the benefit of DP-based size reduction. Another consequence of this random selection could be that OptDebug may localize an entirely different faulty operation, with possibly lesser or higher confidence. This uncertainty exists in any debugging method where the results of automated debugging depend on the chosen execution outcomes under investigation. A possible future work can help OptDebug identify how many p and f outputs to select and which ones to prioritize.

OptDebug's fault localizability thrives in cases when an application is complex and comprises multiple execution paths. In such cases, OptDebug can identify fault-inducing operation with very high confidence as it is more likely that the provenance tags of a passing output have minimal overlap with the provenance tags of a failing output. On the contrary, when an application lacks execution path diversity, OptDebug struggles to isolate fault-inducing operations. When data records flow through a similar set of operations and produce overlapping provenance tags for both passing and failing outcomes, every operation wold have nearly identical suspicious scores.

5 RELATED WORK

Data Provenance. In database systems, data provenance (DP) is the widely used approach for explaining query results and data tracing for debugging purposes. The goal of DP is to find the input data records that contributed toward the generation of another data record [11, 12, 18, 25]. More recently, data provenance techniques have been adapted to support data tracing on data-intensive computing systems. For instance, RAMP [26] and Newt [32] support data provenance on Hadoop-based DISC systems. Both systems enable backward tracing and, therefore, can be used by OptDebug.

However, the two DP techniques provide coarse-grained provenance, as they rely on black-box instrumentation and often produce over-approximated backward traces. Furthermore, they store lineage information in external storage, adding additional runtime overhead and slow provenance query speed. Interlandi et al. overcome these two limitations in Titian by capture data lineages at the data transformation level and by storing lineage tables in in-memory storage [27]. Instead of a traditional two-phase approach (instrumentation followed by provenance query), Smoke uses the specification of data transformation operators to build a data provenance query, eliminating the need of proactively capturing data lineages and thus reducing runtime overhead [38].

Data provenance aims to explain the source of a problem in the *data-space*, *i.e.,* isolating culprit faulty data records as opposed to isolating the problem in *code-space i.e.,* faulty code lines. To our knowledge, OptDebug is the first to fully automate code debugging, going beyond prior work that focused on data debugging only. Since OptDebug leverages DP to perform input reduction, any potential advancements in DP will complement OptDebug, e.g., advancements including but not limited to provenance storage [13], influence [14], input simplification [20], and provenance queries [34].

Spectrum-based Fault Localization (SBFL). In software engineering, a variety of techniques perform fault localization in the *code-space*. The most prominent category is spectrum-based fault localization (SBFL) [42]. SBFL attributes code snippets that participate more often in failing tests than in passing tests have a high chance of causing test failures. More specifically, SBFL captures each individual test's line (or statement) coverage in isolation and contrasts the collective coverage profile of passing and failing tests. The research on SBFL proposes a wide range of ranking methods, including Tarantula [29], Ochiai [9], Barniel [8], and OP2 [35]. Pearson et al. conduct a comprehensive study of which suspicious score methods work best for which type of faults [37].

In the domain of DISC systems, SBFL has very little applicability. SBFL requires running several tests in isolation, which is a reasonable assumption for desktop applications but is prohibitively expensive for DISC applications with long latency for context initialization, data partitioning, etc. SBFL also relies on having an external coverage profiling tool such JaCoCo [4]. Today's coverage tools are designed for single-machine execution only and are not compatible with parallel, distributed execution. OptDebug realizes the idea of SBFL by re-designing underlying support with a novel operation-level taint analysis to overcome the limitation of repetitive execution of each test in isolation.

Taint Analysis. Taint analysis is an active area of research in security and privacy [33, 36]. In software engineering, it has been commonly used for debugging and testing purposes [15, 31]. For example, Penumbra [16] performs faulty-input isolation using taint analysis. It instruments program variables to attach a taint object, similar to OptDebug, to capture the program's data flow. FlowDebug is an influence-based data provenance tool that uses taint analysis to determine the influence-based input and output associations—finding the input that has the biggest impact on an output [40]. Similar to data provenance, both approaches perform debugging in the *data-space* only. Additionally, taint analysis is a costly operation that puts intense pressure on compute and memory needs. OptDebug makes the cost of taint analysis manageable for large-scale DISC applications by carrying out the input simplification step first.

In *code-space* debugging, program slicing isolates a set of statements or variables used to reach a program variable at a specific program point [10, 24, 41]. It comes in two flavors: dynamic and static. In DISC systems, existing program slicing techniques are likely to be ineffective, as program slicing through the underlying runtime (*e.g.,* Spark) will easily over-approximate the slice results. OptDebug overcomes this limitation by staying in the application layer, tagging provenance-tracking to an individual data record.

Debugging Big Data Analytics. Prior work in debugging DISC systems includes interactive debuggers such as Amber [30], BigDebug [21], Dagger [39], and TagSniff [17]. A common goal among these interactive debuggers is to allow a user to inspect the program states of a DISC application, similar to step-through breakpoint and watchpoint debugging in gdb. Such debuggers immensely improve the user's understanding of a DISC application execution. However, unlike OptDebug, interactive debuggers leave it to the user to manually inspect program states to find the root cause in terms of faulty data or faulty code.

6 CONCLUSION

Prior work in DISC application debugging focuses on the problem of *data debugging* only not *code debugging*. To automatically determine the root cause in terms of code lines and API operations, OptDebug proposes a novel operation-level taint analysis to track the history of executed code lines and APIs. It leverages the capability of data provenance for input simplification first and extends taint analysis to a new dimension to tag the history of applied operations, as opposed to the history of information flows (data). OptDebug is highly accurate in detecting code faults (*i.e.,* 100% recall and 86% precision). OptDebug is fast, taking less than half of the original job execution time. To our knowledge, OptDebug is the first fully automated *code debugging* tool that adapts the key idea of spectra-based fault localization to the domain of data-intensive computing applications in a *fast*, *low overhead*, and *highly accurate* manner.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their comments and Venkatesh Emani for his guidance as a shepherd. The participants of this research are in part supported by NSF grants CCF-2106420, CCF-2106404, CNS-2106838, CHS-1956322, CCF-1764077, CCF-1723773, ONR grant N00014-18-1-2037, and Intel CAPA grant.

REFERENCES

[1] 2021. Apache Spark. https://spark.apache.org/.
[2] 2021. Hadoop. http://hadoop.apache.org/.
[3] 2021. Intel HiBench. https://github.com/Intel-bigdata/HiBench.
[4] 2021. JaCoCo. https://jacoco.github.io/.
[5] 2021. Kaggle Datasets. https://www.kaggle.com.
[6] 2021. Pig Mix Benchmark. https://cwiki.apache.org/confluence/display/pig/PigMix/.
[7] 2021. TPC. http://tpc.org/default5.asp/.
[8] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, USA, 88–99. https://doi.org/10.1109/ASE.2009.25
[9] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-Based Fault Localization. *J. Syst. Softw.* 82, 11 (Nov. 2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035
[10] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/93542.93576
[11] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. 2010. Techniques for Efficiently Querying Scientific Workflow Provenance Graphs. In *Proceedings of the 13th International Conference on Extending Database Technology* (Lausanne, Switzerland) *(EDBT '10)*. ACM, New York, NY, USA, 287–298. https://doi.org/10.1145/1739041.1739078
[12] Olivier Biton, Sarah Cohen-Boulakia, Susan B. Davidson, and Carmem S. Hara. 2008. Querying and Managing Provenance Through User Views in Scientific Workflows. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, Washington, DC, USA, 1072–1081. https://doi.org/10.1109/ICDE.2008.4497516
[13] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. 2008. Efficient Provenance Storage. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 993–1006. https://doi.org/10.1145/1376616.1376715
[14] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. 2016. Explaining Outputs in Modern Data Analytics. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1137–1148. https://doi.org/10.14778/2994509.2994530
[15] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/1273463.1273490
[16] James Clause and Alessandro Orso. 2009. Penumbra: Automatically Identifying Failure-relevant Inputs Using Dynamic Tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/1572272.1572301
[17] Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, Zoi Kaoudi, and Saravanan Thirumuruganathan. 2019. TagSniff: Simplified Big Data Debugging for Dataflow Jobs. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 453–464. https://doi.org/10.1145/3357223.3362738
[18] Y. Cui and J. Widom. 2003. Lineage Tracing for General Data Warehouse Transformations. *The VLDB Journal* 12, 1 (May 2003), 41–58. https://doi.org/10.1007/s00778-002-0083-8
[19] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[20] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. 2017. Automated Debugging in Data-Intensive Scalable Computing. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 520–534. https://doi.org/10.1145/3127479.3131624
[21] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 784–795. https://doi.org/10.1145/2884781.2884813
[22] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. 2019. White-Box Testing of Big Data Analytics with Complex User-Defined Functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 290–301. https://doi.org/10.1145/3338906.3338953
[23] Muhammad Ali Gulzar, Siman Wang, and Miryung Kim. 2018. BigSift: Automated Debugging of Big Data Analytics in Data-Intensive Scalable Computing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 863–866. https://doi.org/10.1145/3236024.3264586
[24] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating Faulty Code Using Failure-inducing Chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1101908.1101948
[25] Thomas Heinis and Gustavo Alonso. 2008. Efficient Lineage Tracking for Scientific Workflows. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. ACM, New York, NY, USA, 1007–1018. https://doi.org/10.1145/1376616.1376716
[26] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for generalized map and reduce workflows. In *In Proc. Conference on Innovative Data Systems Research (CIDR)*.
[27] Matteo Interlandi, Ari Ekmekji, Kshitij Shah, Muhammad Ali Gulzar, Sai Deep Tetali, Miryung Kim, Todd Millstein, and Tyson Condie. 2018. Adding Data Provenance Support to Apache Spark. *The VLDB Journal* 27, 5 (Oct. 2018), 595–615. https://doi.org/10.1007/s00778-017-0474-5
[28] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015.

Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 216–227. https://doi.org/10.14778/2850583.2850595

[29] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. ACM, New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397

[30] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. 2020. Amber: A Debuggable Dataflow System Based on the Actor Model. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 740–753. https://doi.org/10.14778/3377369.3377381

[31] Timothy Robert Leek, Graham Z Baker, Ruben Edward Brown, Michael A Zhivich, and RP Lippmann. 2007. *Coverage maximization using dynamic taint tracing.* Technical Report. DTIC Document.

[32] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable lineage capture for debugging DISC analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 17.

[33] W. Masri, A. Podgurski, and D. Leon. 2004. Detecting and debugging insecure information flows. In *15th International Symposium on Software Reliability Engineering*. 198–209. https://doi.org/10.1109/ISSRE.2004.17

[34] Tobias Müller, Benjamin Dietrich, and Torsten Grust. 2018. You Say 'What', i Hear 'where' and 'Why': (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. *Proc. VLDB Endow.* 11, 11 (July 2018), 1536–1549. https://doi.org/10.14778/3236187.3236204

[35] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-Based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 11 (Aug. 2011), 32 pages. https://doi.org/10.1145/2000791.2000795

[36] James Newsome and Dawn Song. 2005. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In In Proceedings of the 12th Network and Distributed Systems Security Symposium*. Citeseer.

[37] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 609–620. https://doi.org/10.1109/ICSE.2017.62

[38] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-Grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 719–732. https://doi.org/10.14778/3199517.3199522

[39] El Kindi Rezig, Lei Cao, Giovanni Simonini, Maxime Schoemans, Samuel Madden, Nan Tang, Mourad Ouzzani, and Michael Stonebraker. 2020. Dagger: A Data (not code) Debugger. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p35-rezig-cidr20.pdf

[40] Jason Teoh, Muhammad Ali Gulzar, and Miryung Kim. 2020. Influence-Based Provenance for Dataflow Applications with Taint Propagation. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 372–386. https://doi.org/10.1145/3419111.3421292

[41] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[42] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[43] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proc. VLDB Endow.* 6, 8 (June 2013), 553–564. https://doi.org/10.14778/2536354.2536356

[44] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 722–733. https://doi.org/10.1145/3324884.3416641