OptDebug: Fault-Inducing Operation Isolation for Dataflow Applications

ACM Symposium of Cloud Computing 2021

Muhammad Ali Gulzar and Miryung Kim

VT VIRGINIA TECH™ UCLA
Prevalence of Big Data Analytics

Use of large-scale data
- Insurance
- Advertisement
- Finance
- ...

Data Processing Systems
- Apache Spark
- Hadoop
- TensorFlow
- Hive
- ...

Big Data Applications
- AWS
- Microsoft Azure
- Big Data Software
- Scala
- Python
- ...

...
Debugging in Traditional Software

- Debugging is interactive and quick.
- Trial and error is feasible. Each execution takes a few milliseconds.
- Direct access to program states and variables.
Debugging in Dataflow Applications

- **Debugging is slow and expensive**, mostly via post mortem logs.
- **Trial and error is time-consuming and expensive**. Each **execution takes a few hours** and expensive compute cycles.
- Due to remote, distributed processing, there is **no easy, direct access to program states and variables**.
Running Example

Calculate the total flying hours for less-than-four hour flights grouped by each departure hour.

Input Dataset

<table>
<thead>
<tr>
<th>Pass ID</th>
<th>Dep Airport</th>
<th>Dep Time</th>
<th>Arr Airport</th>
<th>Arr Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAY993311</td>
<td>CLT</td>
<td>13:15</td>
<td>ORD</td>
<td>15:15</td>
</tr>
<tr>
<td>EWS121311</td>
<td>LAX</td>
<td>10:45</td>
<td>ORD</td>
<td>15:00</td>
</tr>
<tr>
<td>AAQ591783</td>
<td>SJC</td>
<td>03:33</td>
<td>MNN</td>
<td>8:20</td>
</tr>
</tbody>
</table>

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(:)(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) }
  .filter(v => v._2 < 4)
  .reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
  return arr_hr - dep_hr }
```
Running Example

val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split("\,"\)
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) } .filter(v => v._2 < 4) .reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if (arr_hr - dep_hr < 0) { // across midnight
    return arr_hr - dep_hr - 24
  } else {
    return arr_hr - dep_hr
  }

Input Dataset

<table>
<thead>
<tr>
<th>Pass ID</th>
<th>Dep Airport</th>
<th>Dep Time</th>
<th>Arr Airport</th>
<th>Arr Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAY993311</td>
<td>CLT</td>
<td>13:15</td>
<td>ORD</td>
<td>15:15</td>
</tr>
<tr>
<td>EWS121311</td>
<td>LAX</td>
<td>10:45</td>
<td>ORD</td>
<td>15:00</td>
</tr>
<tr>
<td>AAQ591783</td>
<td>SJC</td>
<td>03:33</td>
<td>MNN</td>
<td>8:20</td>
</tr>
</tbody>
</table>
Example: Code Space Debugging is Difficult.

```scala
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff)
}.filter(v => v._2 < 4)
.reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
  return arr_hr - dep_hr }
```

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>2.0</td>
</tr>
<tr>
<td>10</td>
<td>5.75</td>
</tr>
<tr>
<td>03</td>
<td>5.33</td>
</tr>
</tbody>
</table>
Example: Code Space Debugging is Difficult.

Filter

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>2.0</td>
</tr>
<tr>
<td>10</td>
<td>5.75</td>
</tr>
<tr>
<td>03</td>
<td>5.33</td>
</tr>
</tbody>
</table>

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) } .filter(v => v._2 < 4).reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
  return arr_hr - dep_hr }
```
Example: Code Space Debugging is Difficult.

- Data is clean and passes all sanity checks
- Provenance based debugging approaches only debug data-space and not code-space.

```scala
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
val tokens = s.split(",")
val dept_hr = tokens(2).split(":")(0)
val diff = getDiff(tokens(4), tokens(2))
(dept_hr, diff) }
.filter(v => v._2 < 4)
.reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
  return arr_hr - dep_hr }
```
Example: Code Space Debugging is Difficult.

- Data is clean and passes all sanity checks.
- Provenance based debugging approaches only debug data-space and not code-space.

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) }
  .filter(v => v._2 < 4)
  .reduceByKey(_+_) // Calculates the difference between time

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
  return arr_hr - dep_hr }
```

Reduce

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>175080</td>
</tr>
<tr>
<td>20</td>
<td>173460</td>
</tr>
<tr>
<td>23</td>
<td>-222780</td>
</tr>
</tbody>
</table>

Why is Total Flying

How can we precisely detect code (i.e., operations or APIs) responsible for a given suspicious or incorrect result?
Prior Work:
Data Provenance – Data Space Debugging

**Input Dataset**

<table>
<thead>
<tr>
<th>Pass ID</th>
<th>Dep Airport</th>
<th>Dep Time</th>
<th>Arr Airport</th>
<th>Arr Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAY993311</td>
<td>CLT</td>
<td>23:15</td>
<td>ORD</td>
<td>15:15</td>
</tr>
<tr>
<td>EWS121311</td>
<td>LAX</td>
<td>21:45</td>
<td>ORD</td>
<td>04:00</td>
</tr>
<tr>
<td>AAQ591783</td>
<td>SJC</td>
<td>23:33</td>
<td>MNN</td>
<td>5:20</td>
</tr>
</tbody>
</table>

**Map**

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>2.0</td>
</tr>
<tr>
<td>21</td>
<td>5.75</td>
</tr>
<tr>
<td>23</td>
<td>5.33</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Reduce**

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>175080</td>
</tr>
<tr>
<td>20</td>
<td>173460</td>
</tr>
<tr>
<td>23</td>
<td>-222780</td>
</tr>
</tbody>
</table>

Data provenance results of a suspicious output could be large in the order of millions.
Data Provenance – Data Space Debugging

Data Provenance techniques identify culprit records in the data-space. Developers still need to debug in the code-space.

<table>
<thead>
<tr>
<th>Pass ID</th>
<th>Dep Airport</th>
<th>Dep Time</th>
<th>Arr Airport</th>
<th>Arr Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAY993311</td>
<td>CLT</td>
<td>23:15</td>
<td>ORD</td>
<td>15:15</td>
</tr>
<tr>
<td>EWS121311</td>
<td>LAX</td>
<td>21:45</td>
<td>ORD</td>
<td>04:00</td>
</tr>
<tr>
<td>AAQ591783</td>
<td>SJC</td>
<td>23:33</td>
<td>MNN</td>
<td>5:20</td>
</tr>
</tbody>
</table>

**Map**

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>2.0</td>
</tr>
<tr>
<td>21</td>
<td>5.75</td>
</tr>
<tr>
<td>23</td>
<td>5.33</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Filter**

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>2.0</td>
</tr>
<tr>
<td>21</td>
<td>5.75</td>
</tr>
<tr>
<td>23</td>
<td>5.33</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Reduce**

<table>
<thead>
<tr>
<th>Dep Hour</th>
<th>Flying Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>175080</td>
</tr>
<tr>
<td>20</td>
<td>173460</td>
</tr>
<tr>
<td>23</td>
<td>-222780</td>
</tr>
</tbody>
</table>
For traditional software, **spectra-based fault localization** [Jones and Harrold 2002] uses existing test suites to isolate code statements responsible for a test failure.

```python
//Function to convert temperature measurements
def convert( opt: String, value: Float ): Float =
{
    opt match {
        case "KToC" => value - 273.15
        case "CToK" => value + 273.15
        case "FToC" => (value * 5) / 9
        case "CtoF" => (9/5) * value + 32
        _ => value
    }
}
```
Faulty Code Localization

- For traditional software, **spectra-based fault localization** [Jones and Harrold 2002] uses existing test suites to isolate code statements responsible for a test failure.

```python
//Function to convert temperature measurements
def convert(opt: String, value: Float): Float = {
  opt match {
    case "KToC" => value - 273.15
    case "CToK" => value + 273.15
    case "FToC" => (value * 5) / 9
    case "CtoF" => (9/5) * value + 32
    _ => value
  }
}
```

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Suspicious Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Code Coverage
• For traditional software, **spectra-based fault localization** [Jones and Harrold 2002] uses existing test suites to isolate code statements responsible for a test failure.

```python
//Function to convert temperature measurements
def convert( opt: String, value: Float ) : Float =
{
    opt match {
        case "KToC" => value - 273.15
        case "CToK" => value + 273.15
        case "FToC" => (value * 5) / 9
        case "CtoF" => (9/5) * value + 32
        _ => value
    }
}
```

Spectra-Based Fault Localization is not feasible for dataflow applications due to: **large input size, distributed execution, and imprecision**
OptDebug precisely pinpoints the operation and code line number that is responsible for a test failure.
Observation 1: Infeasibility of Code Debugging

Collecting code coverage when running an application on large data is prohibitively expensive.

Input Dataset: 2 billion rows

<table>
<thead>
<tr>
<th>Pass ID</th>
<th>Dep Airport</th>
<th>Dep Time</th>
<th>Arr Airport</th>
<th>Arr Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAY993311</td>
<td>CLT</td>
<td>13:15</td>
<td>ORD</td>
<td>15:15</td>
</tr>
<tr>
<td>EWS121311</td>
<td>LAX</td>
<td>10:45</td>
<td>ORD</td>
<td>15:00</td>
</tr>
<tr>
<td>AAQ591783</td>
<td>SJC</td>
<td>03:33</td>
<td>MNN</td>
<td>8:20</td>
</tr>
</tbody>
</table>

Code Coverage entries ~ 10X of 2 billion rows

val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
    val tokens = s.split(",")
    val dept_hr = tokens(2).split(";")(0)
    val diff = getDiff(tokens(4), tokens(2))
    (dept_hr, diff) }
    .filter(v => v._2 < 4)
    .reduceByKey(_+_)
Insight 1: Test Input Simplification

• Using user-provided test function, we can retrieve simplified passing and failing test input from the dataset.

By reducing data to only culprit input records, we speed-up spectra-based fault code localization on dataflow applications.
Observation 2: Collection of Code Coverage

Traditional coverage tools required system-level modifications to support coverage collection in a distributed setting.

Input Dataset: 2 billion rows

<table>
<thead>
<tr>
<th>Pass ID</th>
<th>Dep Airport</th>
<th>Dep Time</th>
<th>Arr Airport</th>
<th>Arr Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAY993311</td>
<td>CLT</td>
<td>13:15</td>
<td>ORD</td>
<td>15:15</td>
</tr>
<tr>
<td>EWS121311</td>
<td>LAX</td>
<td>10:45</td>
<td>ORD</td>
<td>15:00</td>
</tr>
<tr>
<td>AAQ591783</td>
<td>SJC</td>
<td>03:33</td>
<td>MNN</td>
<td>8:20</td>
</tr>
</tbody>
</table>

```scala
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(" ")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff)
}.filter(v => v._2 < 4)
.reduceByKey(_+_
```

- Requires JVM instrumentation at each node in the cluster.
- Cannot differentiate between application vs framework code
Insight 2: Taint Analysis

- Instead of collecting code coverage at the JVM level, we augment data types with taint containing the history of applied operations.

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) }
.filter(v => v._2 < 4)
.reduceByKey(_+_
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Taint (Line Number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>XAY993311 CLT 13:15 ORD 15:15</td>
<td>[ 3 ]</td>
</tr>
</tbody>
</table>

OptDebug leverages operator overloading and type-inference to capture the code line number at each statement. It is **platform-agnostic**.
Insight 2: Taint Analysis

- Instead of collecting code coverage at the JVM level, we augment data types with taint containing the history of applied operations.

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) }
.filter(v => v._2 < 4)
.reduceByKey(_+_)
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Taint (Line Number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dept_hr</td>
<td>13</td>
<td>[3,4,5]</td>
</tr>
</tbody>
</table>

OptDebug leverages operator overloading and type-inference to capture the code line number at each statement. It is platform-agnostic.
Insight 2: Taint Analysis

- Instead of collecting code coverage at the JVM level, we augment data types with taint containing the history of applied operations.

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split","
  val dept_hr = tokens(2).split":"(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) }
.filter(v => v._2 < 4)
.reduceByKey(_+_)
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Taint (Line Number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dept_hr</td>
<td>13</td>
<td>[3,4,5]</td>
</tr>
<tr>
<td>diff</td>
<td>2</td>
<td>[3,4,5,7,12,13,14,17]</td>
</tr>
</tbody>
</table>

OptDebug leverages operator overloading and type-inference to capture the code line number at each statement. It is platform-agnostic.
Observation 3: Statement Coverage’s Imprecision

Traditional statement coverage only captures line coverage thus incapable of identifying faulty operation.

```scala
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(","
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) }.filter(v => v._2 < 4).reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  // across midnight
  if( arr_hr - dep_hr < 0) return arr_hr - dep_hr - 24
  return arr_hr - dep_hr
}
```
Insight 3: Operation-level Taint Analysis

- OptDebug extends traditional taint analysis to maintain the history of individual operation applied on the data.

```scala
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
  val tokens = s.split(",")
  val dept_hr = tokens(2).split(":")(0)
  val diff = getDiff(tokens(4), tokens(2))
  (dept_hr, diff) }
  .filter(v => v._2 < 4)
  .reduceByKey(_+_
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Taint (Line Number -&gt; Operation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dept_hr</td>
<td>13</td>
<td>[3, 4 -&gt; split, 5 -&gt; split, 5 -&gt; idx ]</td>
</tr>
<tr>
<td>diff</td>
<td>2</td>
<td>[... 16 -&gt; Float.gte, 17 -&gt; Float.minus]</td>
</tr>
</tbody>
</table>

By keeping the history of applied code operations, as opposed to the origin of affected data, OptDebug can precisely identify the faulty operation.
Suspicious Score

- Using Tarantula score (default), OptDebug identifies the operation most likely responsible for a test failure.

<table>
<thead>
<tr>
<th>Output</th>
<th>Taint (Line Number -&gt; Operation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 -222780</td>
<td>[3, 4 -&gt; split, 5 -&gt; split, 5 -&gt; idx ]</td>
</tr>
<tr>
<td>13 173460</td>
<td>[... 16 -&gt; Float.Lt, 17 -&gt; Float.minus]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOC/Operation</th>
<th>Pass Test</th>
<th>Fail Test</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 -&gt; split, 5 -&gt; split, 5 -&gt; idx</td>
<td>[ ]</td>
<td>[ ]</td>
<td>0.5</td>
</tr>
<tr>
<td>16 -&gt; Float.Lt, 17 -&gt; Float.minus</td>
<td>[ ]</td>
<td>[ ]</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Score: 0.5
Using Tarantula score (default), OptDebug identifies the operation most likely responsible for a test failure.

<table>
<thead>
<tr>
<th>Output</th>
<th>Taint (Line Number - &gt; Operation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 -222780</td>
<td>[3, 4 -&gt; split, 5 -&gt; split, 5 -&gt; idx ]</td>
</tr>
<tr>
<td>13 173460</td>
<td>[... 16 -&gt; Float.gte, 17 -&gt; Float.minus]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOC/Operation</th>
<th>Pass Test</th>
<th>Fail Test</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 -&gt; split, 5 -&gt; split, 5 -&gt; idx . . . 16 -&gt; Float.Lt, 17 -&gt; Float.minus . .</td>
<td></td>
<td></td>
<td>0.5 0.5 0.5 . . 0.5 1 0.5 . .</td>
</tr>
</tbody>
</table>

```plaintext
16. // across midnight
17. if( arr_hr - dep_hr < 0) return arr_hr = dep_hr - 24
18. return arr_hr - dep_hr }
```
How well does OptDebug work in Practice?

- We evaluate OptDebug on 6 real-world benchmark programs

- Input Dataset size ranging from 2 GB to 93 GB

- Injected fault inspired by prior study on dataflow application faults reported on Stack overflow and Apache Spark mailing lists

- Comparison against baselines
  - Data Provenance
  - Traditional Spectra-based fault localization
RQ1: 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Row Count</th>
<th>Simplified Input via DP</th>
<th>Known Faults</th>
<th>Detected Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$10^8$</td>
<td>$1.7 \times 10^6$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>$10^7$</td>
<td>$4.5 \times 10^4$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>$10^7$</td>
<td>$2.2 \times 10^5$</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>$10^9$</td>
<td>$1.9 \times 10^5$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P5</td>
<td>$10^7$</td>
<td>210</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P6</td>
<td>$10^9$</td>
<td>$7.0 \times 10^5$</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

OptDebug finds the fault-inducing operation with 86% precision and 100% recall on average.
RQ2: Debugging Time

- We measure the time OptDebug takes to find the fault-inducing operation (i.e., time taken after a given application produces a failing outcome defined via a test predicate).

OptDebug finds the fault-inducing operation with 86% precision and 100% recall on average.
RQ3: Taint Analysis vs. SBFL

- We compare OptDebug’s operation-level taint analysis on running spectra-based fault localization with a simplified input.

OptDebug’s taint analysis on a simplified input is on average 27% faster than applying spectra-based fault localization.
Conclusion

- OptDebug proposes a novel operation-level taint analysis to track the history of executed code lines and APIs to automatically determine the root cause in terms of code lines and API operations.

- OptDebug is a library (jar) that can be imported in any Apache Spark application written in Scala.

https://github.com/maligulzar/OptDebug